The Objective-C Compiler

The Objective-C compiler is based on version 2.7.2 of the GNU C compiler, an ANSI-standard C compiler produced by the Free Software Foundation. The 2.7.2 compiler has been modified and extended as a compiler for the Objective-C language by NeXT Software, Inc. This document describes how to compile a C program using the Objective-C compiler.

This chapter is a modified version of documentation provided by the Free Software Foundation; see the section “Legal Considerations” at the end of this document for important related information.

The following sections describe command options available when compiling a C program, incompatibilities between C as interpreted by this compiler and non-ANSI versions of C, GNU extensions to the C language, and implementation-specific details related to using C.

For a description of the Objective-C language, see *Object-Oriented Programming and the Objective-C Language*.

Which Language?

The C, C++, Objective-C, and Objective-C++ versions of the compiler are integrated; the GNU C compiler can compile programs written in C, C++, or Objective-C. Source code for any of these languages can be ASCII text or Rich Text Format; the preprocessor strips out all RTF directives, leaving only ASCII text for the compiler itself.

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs.

When referring to C++ compilation, the compiler is occasionally referred to as "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context.

We use the name "GNU CC" to refer to the compilation system as a whole, and more specifically to the language-independent part of the compiler. For example, we refer to the optimization options as affecting the behavior of "GNU CC" or sometimes just "the compiler".

G++ is a *compiler*, not merely a preprocessor. G++ builds object code directly from your C++ program source. There is no intermediate C version of the program. (By contrast, for example, some other implementations use a program that generates a C program from your C++ source.) Avoiding an intermediate C representation of the program means that you get better object code, and better debugging information. The GNU debugger, GDB, works with this information in the object code to give you comprehensive C++ source-level editing capabilities. See *The GNU Source-Level Debugger* for more information.

GNU CC Command Options

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command line options that you can use with GNU CC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

See "Compiling C++ Programs" for a summary of special options for compiling C++ programs.

The **gcc** program accepts options and file names as operands. Many options have multiletter names; therefore multiple single-letter options may *not* be grouped: **-dr** is very different from **-d -r**.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify **-L** more than once, the directories are searched in the order specified.

Many options have long names starting with **-f** or with **-W**—for example, **-fforce-mem**, **-fstrength-reduce**, **-Wformat** and so on. Most of these have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. This manual documents only one of these two forms, whichever one is not the default.

Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

*file***.c** C source code which must be preprocessed.

*file***.i** C source code which should not be preprocessed.

*file***.ii** Objective-C++ or C++ source code which should not be preprocessed.

*file***.m** Objective-C source code. Note that you must link with the library **libobjc.a** to make an Objective-C program work.

*file***.mm**
*file***.M** Mixed Objective-C and C++ source code.

*file***.h** C header file (not to be compiled or linked).

*file.***C**
*file***.cc**
*file***.cxx**
*file***.cpp** C++ source code which must be preprocessed. Note that in **.cxx**, the last two letters must both be literally **x**. Likewise, **.C** refers to a literal capital C.

*file***.s** Assembler code.

file.**S** Assembler code which must be preprocessed.

other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the **-x** option:

-x *language*
Specify explicitly the *language* for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next **-x** option. Possible values for *language* are: c objective-c c++ c-header cpp-output c++-cpp-output assembler assembler-with-cpp

-x none Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if **-x** has not been used at all).

If you only want some of the stages of compilation, you can use **-x** (or filename suffixes) to tell **gcc** where to start, and one of the options **-c**, **-S**, or **-E** to say where **gcc** is to stop. Note that some combinations (for example, **-x cpp-output -E**) instruct **gcc** to do nothing at all.

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix **.c**, **.i**, **.s**, etc., with **.o**.

Unrecognized input files, not requiring compilation or assembly, are ignored.

-S Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix **.c**, **.i**, etc., with **.s**.

Input files that don't require compilation are ignored.

-E Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

-o *file* Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

Since only one output file can be specified, it does not make sense to use **-o** when compiling more than one input file, unless you are producing an executable file as output.

If **-o** is not specified, the default is to put an executable file in **a.out**, the object file for *source*.*suffix* in *source***.o**, its assembler file in *source***.s**, and all preprocessed C source on standard output.

-v Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-pipe (Not available on Windows NT) Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

See “Hardware Models and Configurations” for information on the **-arch** flag, which allows you to specify the target platform when compiling on Mach.

Compiling C++ Programs

C++ source files conventionally use one of the suffixes **.C**, **.cc**, **.cpp**, or **.cxx**; preprocessed C++ files use the suffix **.ii**. GNU CC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name **gcc**). Objective-C++ (mixed Objective-C and C++) source files use a **.M** suffix by convention.

However, C++ programs often require class libraries as well as a compiler that understands the C++ language—and under some circumstances, you might want to compile programs from standard input, or otherwise without a suffix that flags them as C++ programs. **g++** is a program that calls GNU CC with the default language set to C++, and automatically specifies linking against the GNU class library libg++. On many systems (but not Windows NT), the script **g++** is also installed with the name **c++**.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See “Options Controlling C Dialect”, for explanations of options for languages related to C. See “Options Controlling C++ Dialect”, for explanations of options that are meaningful only for C++ programs.

Options Controlling C Dialect

The following options control the dialect of C (or languages derived from C, such as C++ and Objective-C) that the compiler accepts:

-ansi Support all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the **asm**, **inline** and **typeof** keywords, and predefined macros such as **unix** and **vax** that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, disallows **$** as part of identifiers, and disables recognition of C++ style // comments.

The alternate keywords **__asm__**, **__extension__**, **__inline__** and **__typeof__** continue to work despite **-ansi**. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with **-ansi**. Alternate predefined macros such as **__unix__** and **__vax__** are also available, with or without **-ansi**.

The **-ansi** option does not cause non-ANSI programs to be rejected gratuitously. For that, **-pedantic** is required in addition to **-ansi**. See “Options to Request or Suppress Warnings” for more information.

The macro **__STRICT_ANSI__** is predefined when the **-ansi** option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn’t call for; this is to avoid interfering with any programs that might use these names for other things.

The functions **alloca**, **abort**, **exit**, and **_exit** are not builtin functions when **-ansi** is used.

-ObjC (Not supported on PDO platforms) Compile a source file that contains Objective-C language code (the file can have either a **.c** or a **.m** extension).

-fno-asm Do not recognize **asm**, **inline** or **typeof** as a keyword, so that code can use these words as identifiers. You can use the keywords **__asm__**, **__inline__** and **__typeof__** instead. **-ansi** implies **-fno-asm**.

In C++, this switch only affects the **typeof** keyword, since **asm** and **inline** are standard keywords. You may want to use the **-fno-gnu-keywords** flag instead, as it also disables the other, C++-specific, extension keywords such as **headof**.

-fno-builtin

Don’t recognize builtin functions that do not begin with two leading underscores. Currently, the functions affected include **abort**, **abs**, **alloca**, **cos**, **exit**, **fabs**, **ffs**, **labs**, **memcmp**, **memcpy**, **sin**, **sqrt**, **strcmp**, **strcpy**, and **strlen**.

GCC normally generates special code to handle certain builtin functions more efficiently; for instance, calls to **alloca** may become single instructions that adjust the stack directly, and calls to **memcpy** may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

The **-ansi** option prevents **alloca** and **ffs** from being builtin functions, since these functions do not have an ANSI standard meaning.

-trigraphs Support ANSI C trigraphs. You don’t want to know about this brain-damage. The **-ansi** option implies **-trigraphs**.

-traditional Attempt to support some aspects of traditional C compilers. Specifically:

- All **extern** declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The newer keywords **typeof**, **inline**, **signed**, **const** and **volatile** are not recognized. (You can still use the alternative keywords such as **__typeof__**, **__inline__**, and so on.)
- Comparisons between pointers and integers are always allowed.
- Integer types **unsigned short** and **unsigned char** promote to **unsigned int**.
- Out-of-range floating point literals are not an error.
- Certain constructs which ANSI regards as a single invalid preprocessing number, such as **0xe-0xd**, are treated as expressions instead.
- String “constants” are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of **-fwritable-strings**.)
- All automatic variables not declared **register** are preserved by **longjmp**. Ordinarily, GNU C follows ANSI C: automatic variables not declared **volatile** may be clobbered.

- The character escape sequences **\x** and **\a** evaluate as the literal characters **x** and **a** respectively. Without **-traditional**, **\x** is a prefix for the hexadecimal representation of a character, and **\a** produces a bell.
- In C++ programs, assignment to **this** is permitted with **-traditional**. (The option **-fthis-is-variable** also has this effect.)
- You may wish to use **-fno-builtin** as well as **-traditional** if your program uses names that are normally GNU C builtin functions for other purposes of its own.
- You cannot use **-traditional** if you include any header files that rely on ANSI C features. Some vendors are starting to ship systems with ANSI C header files and you cannot use **-traditional** on such systems to compile files that include any system headers.

In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

In preprocessing directive, the **#** symbol must appear as the first character of a line.

In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

The predefined macro **__STDC__** is not defined when you use **-traditional**, but **__GNUC__** is (since the GNU extensions which **__GNUC__** indicates are not affected by **-traditional**). If you need to write header files that work differently depending on whether **-traditional** is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers. The predefined macro **__STDC_VERSION__** is also not defined when you use **-traditional**. See “Standard Predefined Macros” in *The GNU C Preprocessor* for more discussion of these and other predefined macros.

The preprocessor considers a string constant to end at a newline (unless the newline is escaped with \). (Without **-traditional**, string constants can contain the newline character as typed.)

-traditional-cpp

Controls which preprocessor is used. The default is **cpp_precomp**; if you specify this flag, the standard GNU **cpp** will be used instead.

-fcond-mismatch
Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

-funsigned-char
Let the type **char** be unsigned, like **unsigned char**.

Each kind of machine has a default for what **char** should be. It is either like **unsigned char** by default or like **signed char** by default.

Ideally, a portable program should always use **signed char** or **unsigned char** when it depends on the signedness of an object. But many programs have been written to use plain **char** and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type **char** is always a distinct type from each of **signed char** or **unsigned char**, even though its behavior is always just like one of those two.

-fsigned-char
Let the type **char** be signed, like **signed char**.

Note that this is equivalent to **-fno-unsigned-char**, which is the negative form of **-funsigned-char**. Likewise, the option **-fno-signed-char** is equivalent to **-funsigned-char**.

-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields
These options control whether a bitfield is signed or unsigned, when the declaration does not use either **signed** or **unsigned**. By default, such a bitfield is signed, because this is consistent: the basic integer types such as **int** are signed types.

However, when **-traditional** is used, bitfields are all unsigned no matter what.

-fwritable-strings
Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. The option **-traditional** also has this effect.

Writing into string constants is a very bad idea; “constants” should be constant.

-fallow-single-precision
Do not promote single precision math operations to double precision, even when compiling with **-traditional**.

Traditional K&R C promotes all floating point operations to double precision, regardless of the sizes of the operands. On the architecture for which you are compiling, single precision may be faster than double precision. If you must use **-traditional**, but want to use single precision operations when the operands are single precision, use this option. This option has no effect when compiling with ANSI or GNU C conventions (the default).

Options Controlling C++ Dialect

This section describes the command-line options that are only meaningful for C++ programs; but you can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file **firstClass.C** like this:

```
g++ -g -felide-constructors -O -c firstClass.C
```

In this example, only **-felide-constructors** is an option meant only for C++ programs; you can use the other options with any language supported by GNU CC.

Here is a list of options that are *only* for compiling C++ programs:

-ObjC++ (Not supported on PDO platforms) Overrides the path extension so that file contents are interpreted as C++ or Objective-C++ language code.

-fno-access-control
Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.

-fall-virtual

Treat all possible member functions as virtual, implicitly. All member functions (except for constructor functions and **new** or **delete** member operators) are treated as virtual functions of the class where they appear.

This does not mean that all calls to these member functions will be made through the internal table of virtual functions. Under some circumstances, the compiler can determine that a call to a given virtual function can be made directly; in these cases the calls are direct in any case.

-fcheck-new

Check that the pointer returned by **operator new** is non-null before attempting to modify the storage allocated. The current Working Paper requires that **operator new** never return a null pointer, so this check is normally unnecessary.

-fconserve-space

Put uninitialized or runtime-initialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. If you compile with this flag and your program mysteriously crashes after **main()** has completed, you may have an object that is being destroyed twice because two definitions were merged.

-fdollars-in-identifiers Accept **$** in identifiers. You can also explicitly prohibit use of **$** with the option **-fno-dollars-in-identifiers**. (GNU C++ allows **$** by default on some target systems but not others.) Traditional C allowed the character **$** to form part of identifiers. However, ANSI C and C++ forbid **$** in identifiers.

-fenum-int-equiv

Anachronistically permit implicit conversion of **int** to enumeration types. Current C++ allows conversion of **enum** to **int**, but not the other way around.

-fexternal-templates

Cause template instantiations to obey **#pragma interface** and implementation; template instances are emitted or not according to the location of the template definition. See “Where’s the Template?” for more information.

-falt-external-templates

Similar to -fexternal-templates, but template instances are emitted or not according to the place where they are first instantiated. See “Where’s the Template?” for more information.

-ffor-scope -fno-for-scope

If -ffor-scope is specified, the scope of variables declared in a for-init-statement is limited to the **for** loop itself, as specified by the draft C++ standard. If -fno-for-scope is specified, the scope of variables declared in a for-init-statement extends to the end of the enclosing scope, as was the case in old versions of gcc, and other (traditional) implementations of C++.

The default if neither flag is given to follow the standard, but to allow and give a warning for old-style code that would otherwise be invalid, or have different behavior.

-fno-gnu-keywords

Do not recognize **classof**, **headof**, **signature**, **sigof** or **typeof** as a keyword, so that code can use these words as identifiers. You can use the keywords **__classof__**, **__headof__**, **__signature__**, **__sigof__**, and **__typeof__** instead. **-ansi** implies **-fno-gnu-keywords**.

-fno-implicit-templates

Never emit code for templates which are instantiated implicitly (that is, by use); only emit code for explicit instantiations. See “Where’s the Template?” for more information.

-fhandle-signatures

Recognize the **signature** and **sigof** keywords for specifying abstract types. The default (**-fno-handle-signatures**) is not to recognize them. See “Type Abstraction using Signatures”.

-fhuge-objects

Support virtual function calls for objects that exceed the size representable by a **short int**. Users should not use this flag by default; if you need to use it, the compiler will tell you so. If you compile any of your code with this flag, you must compile *all* of your code with this flag (including libg++, if you use it).

This flag is not useful when compiling with **-fvtable-thunks**.

-fno-implement-inlines

To save space, do not emit out-of-line copies of inline functions controlled by **#pragma implementation**. This will cause linker errors if these functions are not inlined everywhere they are called.

-fmemoize-lookups -fsave-memoized

Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.

The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this:

```
cout << “This “ << p << “ has “ << n << “ legs.\n”;
```

makes six passes through all three steps. By using a software cache, a “hit” significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms which must be implemented, and so incurs its own overhead. **-fmemoize-lookups** enables the software cache.

Because access privileges (visibility) to members and member functions may differ from one function context to the next, G++ may need to flush the cache. With the **-fmemoize-lookups** flag, the cache is flushed after every function that is compiled. The **-fsave-memoized** flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

The code that implements these flags has rotted; you should probably avoid using them.

-fstrict-prototype

Within an **extern “C”** linkage specification, treat a function declaration with no arguments, such as **int foo ();**, as declaring the function to take no arguments. Normally, such a declaration means that the function **foo** can take any combination of arguments, as in C. **-pedantic** implies **-fstrict-prototype** unless overridden with **-fno-strict-prototype**.

This flag no longer affects declarations with C++ linkage.

-fno-nonnull-objects

Don’t assume that a reference is initialized to refer to a valid object. Although the current C++ Working Paper prohibits null references, some old code may rely on them, and you can use **-fno-nonnull-objects** to turn on checking.

At the moment, the compiler only does this checking for conversions to virtual base classes.

-foperator-names
Recognize the operator name keywords **and**, **bitand**, **bitor**, **compl**, **not**, **or** and **xor** as synonyms for the symbols they refer to. **-ansi** implies **-foperator-names**.

-fthis-is-variable
Permit assignment to **this**. The incorporation of user-defined free store management into C++ has made assignment to **this** an anachronism. Therefore, by default it is invalid to assign to **this** within a class member function; that is, GNU C++ treats **this** in a member function of class **X** as a non-lvalue of type **X ***. However, for backwards compatibility, you can make it valid with **-fthis-is-variable**.

-fvtable-thunks
Use **thunks** to implement the virtual function dispatch table (**vtable**). The traditional (cfront-style) approach to implementing vtables was to store a pointer to the function and two offsets for adjusting the **this** pointer at the call site. Newer implementations store a single pointer to a **thunk** function which does any necessary adjustment and then calls the target function.

This option also enables a heuristic for controlling emission of vtables; if a class has any non-inline virtual functions, the vtable will be emitted in the translation unit containing the first one of those.

-nostdinc++
Do not search for header files in the standard directories specific to C++, but do still search the other standard directories. (This option is used when building libg++.)

-traditional
For C++ programs (in addition to the effects that apply to both C and C++), this has the same effect as **-fthis-is-variable**. See “Options Controlling C Dialect”.

In addition, these optimization, warning, and code generation options have meanings only for C++ programs:

-fno-default-inline
Do not assume **inline** for functions defined inside a class scope. See “Options That Control Optimization”.

-Wenum-clash -Woverloaded-virtual -Wtemplate-debugging
Warnings that apply only to C++ programs. See “Options to Request or Suppress Warnings”.

+e*n* Control how virtual function definitions are used, in a fashion compatible with **cfront** 1.x. See “Options for Code Generation Conventions”.

Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning **-W**, for example **-Wimplicit** to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning **-Wno-** to turn off warnings; for example, **-Wno-implicit**. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

-fsyntax-only
Check the code for syntax errors, but don’t do anything beyond that.

-pedantic Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require **-ansi**). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

-pedantic does not cause warning messages for use of the alternate keywords whose names begin and end with **__**. Pedantic warnings are also disabled in the expression that follows **__extension__**. However, only system header files should use these escape routes; application programs should avoid them. See “Alternate Keywords”.

This option is not intended to be useful; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard.

Some users try to use **-pedantic** to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all—only those for which ANSI C *requires* a diagnostic.

A feature to report any failure to conform to ANSI C might be useful in some instances, but would require considerable additional work and would be quite different from **-pedantic**. We recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

-pedantic-errors
Like **-pedantic**, except that errors are produced rather than warnings.

-w Inhibit all warning messages.

-Wno-import
Inhibit warning messages about the use of **#import**.

-Wno-precomp
Inhibit warning messages reltaing to not being able to use precompiled headers.

-Wchar-subscripts
Warn if an array subscript has type **char**. This is a common cause of error, as programmers often forget that this type is signed on some machines.

-Wcomment
Warn whenever a comment-start sequence **/*** appears in a comment.

-Wformat Check calls to **printf** and **scanf**, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

-Wimplicit
Warn whenever a function or parameter is implicitly declared.

-Wparentheses
Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

-Wreturn-type
Warn whenever a function is defined with a return-type that defaults to **int**. Also warn about any **return** statement with no return-value in a function whose return-type is not **void**.

-Wstyle Warn when assignments are used as conditionals in **if**, **for**, and **while** statements. For example, consider the following line of code:

```
if (i = foo()) { ... }
```

The warning suggests an extra set of parenthesis around the assignment, like:

```
if ((i = foo())) { ... }
```

The intent behind this warning is to catch situations where you really meant to test for equivalence (==) and not assignment (=).

-Wswitch Warn whenever a **switch** statement has an index of enumeral type and lacks a **case** for one or more of the named codes of that enumeration. (The presence of a **default** label prevents this warning.) **case** labels outside the enumeration range also provoke warnings when this option is used.

-Wtrigraphs
Warn if any trigraphs are encountered (assuming they are enabled).

-Wunused Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used.

To suppress this warning for an expression, simply cast it to void. For unused variables and parameters, use the **unused** attribute (see “Specifying Attributes of Variables”).

-Wuninitialized
An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don’t specify -O, you simply won’t get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared volatile, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
    int x;
    switch (y) {
    case 1:
       x = 1;
       break;
    case 2:
       x = 4;
       break;
    case 3:
       x = 5;
    }
    foo (x);
}
```

If the value of y is always 1, 2 or 3, then x is always initialized, but GNU CC doesn’t know this. Here is another common case:

```
{
    int save_y;
    if (change_y)
       save_y = y, y = new_y;
    ...
    if (change_y)
       y = save_y;
}
```

This has no bug because save_y is used only if it is set.

Some spurious warnings can be avoided if you declare all the functions you use that never return as noreturn. See “Declaring Attributes of Functions”.

-Wenum-clash
Warn about conversion between different enumeration types. (C++ only).

-Wreorder (C++ only)
Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

```
struct A {
    int i;
    int j;
    A(): j (0), i (1) { }
};
```

Here the compiler will warn that the member initializers for i and j will be rearranged to match the declaration order of the members.

-Wtemplate-debugging
When using templates in a C++ program, warn if debugging is not yet fully available (C++ only).

-Wall All of the above **-W** options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining **-W..** options are not implied by **-Wall** because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

-Wmost
-W Print extra warning messages for these events:

- A nonvolatile automatic variable might be changed by a call to **longjmp**. These warnings as well are possible only in optimizing compilation.

  The compiler sees only the calls to **setjmp**. It cannot know where **longjmp** will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because **longjmp** cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a) {
    if (a > 0)
       return a;
}
```

- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as **x[i,j]** will cause a warning, but **x[(void)i,j]** will not.

- An unsigned value is compared against zero with < or <=.

- A comparison like **x<=y<=z** appears; this is equivalent to **(x<=y ? 1 : 0) <= z**, which is a different interpretation from that of ordinary mathematical notation.

- Storage-class specifiers like **static** are not the first things in a declaration. According to the C Standard, this usage is obsolescent.

- If **-Wall** or **-Wunused** is also specified, warn about unused arguments.

- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for **x.h**:

```
struct s {
    int f, g;
};
struct t {
    struct s h;
    int i;
};
struct t x = { 1, 2, 3 };
```

-Wtraditional

Warn about certain constructs that behave differently in traditional and ANSI C.

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.

- A function declared external in one block and then used after the end of the block.

- A **switch** statement has an operand of type **long**.

-Wshadow
Warn whenever a local variable shadows another local variable.

-Wid-clash-*len*
Warn whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

-Wlarger-than-*len*
Warn whenever an object of larger than *len* bytes is defined.

-Wpointer-arith
Warn about anything that depends on the "size of" a function type or of **void**. GNU C assigns these types a size of 1, for convenience in calculations with **void *** pointers and pointers to functions.

-Wbad-function-cast
Warn whenever a function call is cast to a non-matching type. For example, warn if **int malloc()** is cast to **anything ***.

-Wcast-qual
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a **const char *** is cast to an ordinary **char ***.

-Wcast-align
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a **char *** is cast to an **int *** on machines where integers can only be accessed at two- or four-byte boundaries.

-Wwrite-strings
Give string constants the type **const char[LENGTH]** so that copying the address of one into a non-**const char *** pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using **const** in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make **-Wall** request these warnings.

-Wconversion

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment **x = -1** if **x** is unsigned. But do not warn about explicit casts like **(unsigned) -1**.

-Waggregate-return

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

-Wstrict-prototypes

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

-Wmissing-prototypes

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

-Wmissing-declarations

Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files.

-Wredundant-decls

Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

-Wnested-externs

Warn if an **extern** declaration is encountered within an function.

-Winline

Warn if a function can not be inlined, and either it was declared as inline, or else the **-finline-functions** option was given.

-Woverloaded-virtual

Warn when a derived class function declaration may be an error in defining a virtual function (C++ only). In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. With this option, the compiler warns when you define a function with the same name as a virtual function, but with a type signature that does not match any declarations from the base class.

-Wsynth

Warn when g++'s synthesis behavior does not match that of cfront. For instance:

```
struct A {
    operator int ();
    A& operator = (int);
};
main () {
    A a,b;
    a = b;
}
```

In this example, g++ will synthesize a default **A& operator = (const A&);**, while cfront will use the user-defined **operator =**.

-Werror Make all warnings into errors.

Options for Debugging Your Program or GNU CC

GNU CC has various special options that are used for debugging either your program or GCC:

-g Produce debugging information in the operating system's native format. GDB can work with this debugging information.

On most systems that use stabs format, **-g** enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use **-gstabs+**, **-gstabs**, or, on Windows NT, **-gcodeview** or **-gcodeview+** (see below).

Unlike most other C compilers, GNU CC allows you to use **-g** with **-O**. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GNU CC is generated with the capability for more than one debugging format.

-gcodeview Produce debugging information in the stabs format, including Codeview extensions if at all possible.

-ggdb Produce debugging information in the stabs format, including GDB extensions if at all possible.

-gstabs Produce debugging information in stabs format, without GDB extensions.

-gstabs+ Produce debugging information in stabs format, using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

-g*level*
-ggdb*level*
-gstabs*level*
-gcodeview*level* (on Windows NT only)
Request debugging information and also use *level* to specify how much information. The default level is 2.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use **-g3**.

-p (Not available on Windows NT) Generate extra code to write profile information suitable for the analysis program **prof**. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-pg (Not available on Windows NT) Generate extra code to write profile information suitable for the analysis program **gprof**. You must use this option when compiling the source files you want data about, and you must also use it when linking.

-a Generate extra code to write profile information for basic blocks, which will record the number of times each basic block is executed, the basic block start address, and the function name containing the basic block. If **-g** is used, the line number and filename of the start of the basic block will also be recorded. If not overridden by the machine description, the default action is to append to the text file **bb.out**.

This data could be analyzed by a program like tcov. Note, however, that the format of the data is not what tcov expects. Eventually GNU gprof should be extended to process this data.

-d*letters*

Says to make debugging dumps during compilation at times specified by *letters*. This is used for debugging the compiler. The file names for most of the dumps are made by appending a word to the source file name (e.g. **foo.c.rtl** or **foo.c.jump**). Here are the possible letters for use in *letters*, and their meanings:

- **y** Dump debugging information during parsing, to standard error.
- **r** Dump after RTL generation, to *file***.rtl**.
- **x** Just generate RTL for a function instead of compiling it. Usually used with **r**.
- **j** Dump after first jump optimization, to *file***.jump**.
- **s** Dump after CSE (including the jump optimization that sometimes follows CSE), to *file***.cse**.
- **L** Dump after loop optimization, to *file***.loop**.
- **t** Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to *file***.cse2**.
- **f** Dump after flow analysis, to *file***.flow**.
- **c** Dump after instruction combination, to the file *file***.combine**.
- **S** Dump after the first instruction scheduling pass, to *file***.sched**.
- **l** Dump after local register allocation, to *file***.lreg**.

- **g** Dump after global register allocation, to *file***.greg**.
- **R** Dump after the second instruction scheduling pass, to *file***.sched2**.
- **J** Dump after last jump optimization, to *file***.jump2**.
- **d** Dump after delayed branch scheduling, to *file***.dbr**.
- **k** Dump after conversion from registers to stack, to *file***.stack**.
- **a** Produce all the dumps listed above.
- **m** Print statistics on memory usage, at the end of the run, to standard error.
- **p** Annotate the assembler output with a comment indicating which pattern and alternative was used.

In addition, the following letters are preprocessor flags and can only be used with **-traditional-cpp**:

- **M** Dump all macro definitions, at the end of preprocessing, and write no output.
- **N** Dump all macro names, at the end of preprocessing.
- **D** Dump all macro definitions, at the end of preprocessing, in addition to normal output.

-fpretend-float
When running a cross-compiler, pretend that the target machine uses the same floating point format as the host machine. This causes incorrect output of the actual floating constants, but the actual instruction sequence will probably be the same as GNU CC would make when running on the target machine.

-save-temps
Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling **foo.c** with **-c -save-temps** would produce files **foo.i** and **foo.s**, as well as **foo.o**.

-print-file-name=*library*
Print the full absolute name of the library file *library* that would be used when linking—and don't do anything else. With this option, GNU CC does not compile or link anything; it just prints the file name.

-print-prog-name=*program*
Like **-print-file-name**, but searches for a program such as **cpp**.

-print-libgcc-file-name

Same as **-print-file-name=libgcc.a**.

This is useful when you use **-nostdlib** or **-nodefaultlibs** but you do want to link with **libgcc.a**. You can do

gcc -nostdlib FILES... gcc -print-libgcc-file-name

-print-search-dirs

Print the name of the configured installation directory and a list of program and library directories gcc will search—and don't do anything else.

This is useful when gcc prints the error message **installation problem, cannot exec cpp: No such file or directory**. To resolve this you either need to put **cpp** and the other compiler components where gcc expects to find them, or you can set the environment variable **GCC_EXEC_PREFIX** to the directory where you installed them. Don't forget the trailing /. See "Environment Variables Affecting GNU CC".

Options That Control Optimization

These options control various sorts of optimizations:

-O -O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without **-O**, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without **-O**, the compiler only allocates variables declared **register** in registers. The resulting compiled code is a little worse than produced by PCC without **-O**.

With **-O**, the compiler tries to reduce code size and execution time.

When you specify **-O**, the compiler turns on **-fthread-jumps** and **-fdefer-pop** on all machines. The compiler turns on **-fdelayed-branch** on machines that have delay slots, and **-fomit-frame-pointer** on machines that can support debugging even without a frame pointer. On some machines the compiler also turns on other flags.

-O2 Optimize even more. GNU CC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify **-O2**. As compared to **-O**, this option increases both compilation time and the performance of the generated code.

-O2 turns on all optional optimizations except for loop unrolling and function inlining. It also turns on the **-fforce-mem** option on all machines and frame pointer elimination on machines where doing so does not interfere with debugging.

-O3 Optimize yet more. **-O3** turns on all optimizations specified by **-O2** and also turns on the **inline-functions** option.

-O0 Do not optimize.

If you use multiple **-O** options, with or without level numbers, the last such option is the one that is effective.

Options of the form **-f***flag* specify machine-independent flags. Most flags have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing **no-** or adding it.

-ffloat-store

Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.

The floating-point hardware in the **i386** and **m68k** architectures is IEEE-compliant. However, they normally deliver results to extended precision (which the IEEE Standard allows), whereas on other platforms—such as **hppa** and **sparc**—results can be delivered to any supported precision. The **-ffppc** flag is used to make arithmetic behave more like that on other platforms. **-ffloat-store** will achieve this purpose in many cases, but certainly not all, and at a fairly high cost in terms of performance. **-ffppc** will achieve this purpose on **m68k** in most cases at a much lower cost in terms of performance. On **i386**, **-ffppc** will achieve this purpose more often than **-ffloat-store**, in most cases at a much lower cost in terms of performance. **-ffloat-store** is likely to achieve this purpose in cases where **-ffppc** doesn't.

-ffppc (Not available on Windows NT) Ensures that generated code is fully IEEE compliant. Use this option instead of **-ffloat-store**. See the explanation of **-ffloat-store**, above, for more information.

-fno-default-inline

Do not make member functions inline by default merely because they are defined inside the class scope (C++ only). Otherwise, when you specify **-O**, member functions defined inside class scope are compiled inline by default; that is, you don't need to add **inline** in front of the member function name.

-fno-defer-pop

Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

-fforce-mem

Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. The **-O2** option turns on this option.

-fforce-addr

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as **-fforce-mem** may.

-fomit-frame-pointer

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. *It also makes debugging impossible on some machines.*

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro **FRAME_POINTER_REQUIRED** controls whether a target machine supports this flag.

-fno-inline

Don't pay attention to the **inline** keyword. Normally this option is used to keep the compiler from expanding any functions inline. Note that if you are not optimizing, no functions can be expanded inline.

-finline-functions

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared **static**, then the function is normally not output as assembler code in its own right.

-fkeep-inline-functions

Even if all calls to a given function are integrated, and the function is declared **static**, nevertheless output a separate run-time callable version of the function.

-fno-function-cse

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

-ffast-math

This option allows GCC to violate some ANSI or IEEE rules and/or specifications in the interest of optimizing code for speed. For example, it allows the compiler to assume arguments to the **sqrt** function are non-negative numbers and that no floating-point values are NaNs.

This option should never be turned on by any **-O** option since it can result in incorrect output for programs which depend on an exact implementation of IEEE or ANSI rules/specifications for math functions.

The following options control specific optimizations. The **-O2** option turns on all of these optimizations except **-funroll-loops** and **-funroll-all-loops**. On most machines, the **-O** option turns on the **-fthread-jumps** and **-fdelayed-branch** options, but specific machines may handle it differently.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

-fstrength-reduce
Perform the optimizations of loop strength reduction and elimination of iteration variables.

-fthread-jumps
Perform optimizations where the compiler checks to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

-fcse-follow-jumps
In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an **if** statement with an **else** clause, CSE will follow the jump when the condition tested is false.

-fcse-skip-blocks
This is similar to **-fcse-follow-jumps**, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple **if** statement with no else clause, **-fcse-skip-blocks** causes CSE to follow the jump around the body of the **if**.

-frerun-cse-after-loop
Re-run common subexpression elimination after loop optimizations has been performed.

-fexpensive-optimizations
Perform a number of minor optimizations that are relatively expensive.

-fdelayed-branch
(Not available on Windows NT) If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

-fschedule-insns

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

-fschedule-insns2

Similar to **-fschedule-insns**, but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

-fcaller-saves

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

-funroll-loops

Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. **-funroll-loop** implies both **-fstrength-reduce** and **-frerun-cse-after-loop**.

-funroll-all-loops

Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. **-funroll-all-loops** implies **-fstrength-reduce** as well as **-frerun-cse-after-loop**.

-fno-peephole

Disable any machine-specific peephole optimizations.

Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the **-E** option, nothing is done except preprocessing. Some of these options make sense only together with **-E** because they cause the preprocessor output to be unsuitable for actual compilation.

-framework *framework-name*

Search the framework named *framework-name* for header files. The directories searched include **/LocalLibrary/Frameworks** and **/NextLibrary/Frameworks** (both are prefaced by $NEXT_ROOT on Windows NT).

-include *file*

Process *file* as input before processing the regular input file. In effect, the contents of *file* are compiled first. Any **-D** and **-U** options on the command line are always processed before **-include** *file*, regardless of the order in which they are written. All the **-include** and **-imacros** options are processed in the order in which they are written.

-imacros *file*

Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of **-imacros** *file* is to make the macros defined in *file* available for use in the main input.

Any **-D** and **-U** options on the command line are always processed before **-imacros** *file*, regardless of the order in which they are written. All the **-include** and **-imacros** options are processed in the order in which they are written.

-idirafter *dir*

Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that **-I** adds to).

-iprefix *prefix*

Specify *prefix* as the prefix for subsequent **-iwithprefix** options.

-iwithprefix *dir*

Add a directory to the second include path. The directory's name is made by concatenating *prefix* and *dir*, where *prefix* was specified previously with **-iprefix**. If you have not specified a prefix yet, the directory containing the installed passes of the compiler is used as the default.

-iwithprefixbefore *dir*

Add a directory to the main include path. The directory's name is made by concatenating *prefix* and *dir*, as in the case of **-iwithprefix**.

-isystem *dir*

Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

-nostdinc

Do not search the standard system directories for header files. Only the directories you have specified with **-I** options (and the current directory, if appropriate) are searched. See "Options for Directory Search" for information on **-I**.

By using both **-nostdinc** and **-I-**, you can limit the include-file search path to only those directories you specify explicitly.

-undef Do not predefine any nonstandard macros. (Including architecture flags).

-E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

-C Tell the preprocessor not to discard comments. Used with the **-E** option.

-P Tell the preprocessor not to generate **#line** directives. Used with the **-E** option.

-M Tell the preprocessor to output a rule suitable for **make** describing the dependencies of each object file. For each source file, the preprocessor outputs one **make**-rule whose target is the object file name for that source file and whose dependencies are all the **#include** header files it uses. This rule may be a single line or may be continued with \-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

-M implies **-E**.

Another way to specify output of a **make** rule is by setting the environment variable **DEPENDENCIES_OUTPUT** (see "Environment Variables Affecting GNU CC").

-MM Like **-M** but the output mentions only the user header files included with **#include "***file***"**. System header files included with **#include <***file***>** are omitted.

-MD Like **-M** but the dependency information is written to a file made by replacing ".c" with ".d" at the end of the input file names. This is in addition to compiling the file as specified—**-MD** does not inhibit ordinary compilation the way **-M** does.

In Mach, you can use the utility **md** to merge multiple dependency files into a single dependency file suitable for using with the **make** command.

-MMD Like **-MD** except mention only user header files, not system header files.

-MG Treat missing header files as generated files and assume they live in the same directory as the source file. If you specify **-MG**, you must also specify either **-M** or **-MM**. **-MG** is not supported with **-MD** or **-MMD**. This flag is not supported on Mach.

-H Print the name of each header file used, in addition to other normal activities. This flag is not supported on Mach.

-A*question*(*answer*)
Assert the answer *answer* for *question*, in case it is tested with a preprocessing conditional such as **#if #***question*(*answer*). **-A-** disables the standard assertions that normally describe the target machine. This flag is not supported on Mach.

-D*macro*
Define macro *macro* with the string **1** as its definition.

-D*macro=defn*
Define macro *macro* as *defn*. All instances of **-D** on the command line are processed before any **-U** options.

-U*macro*
Undefine macro *macro*. **-U** options are evaluated after all **-D** options, but before any **-include** and **-imacros** options.

-dM Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the **-E** option. This flag is not supported on Mach.

-dD Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output. This flag is not supported on Mach.

-dN Like **-dD** except that the macro arguments and contents are omitted. Only **#define** *name* is included in the output. This flag is not supported on Mach.

-trigraphs
Support ANSI C trigraphs. The **-ansi** option also has this effect.

-Wp,*option*
Pass *option* as an option to the preprocessor. If *option* contains commas, it is split into multiple options at the commas.

Passing Options to the Assembler

You can pass options to the assembler.

-Wa,*option*

Pass *option* as an option to the assembler. If *option* contains commas, it is split into multiple options at the commas.

Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

object-file-name

A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

-c -S -E

If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See “Options Controlling the Kind of Output”.

-l*library*

Search the library named *library* when linking.

It makes a difference where in the command you write this option; the linker searches processes libraries and object files in the order they are specified. Thus, **foo.o -lz bar.o** searches library **z** after file **foo.o** but before **bar.o**. If **bar.o** refers to functions in **z**, those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named **lib***library*.a. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with **-L**.

Normally the files found this way are library files—archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an **-l** option and specifying a file name is that **-l** surrounds *library* with **lib** and **.a** and searches several directories.

-framework *framework-name*
Search the framework named *framework-name* when linking.

The linker searches a standard list of directories for the framework. The linker then uses this file as if it had been specified precisely by name.

The directories searched include **/LocalLibrary/Frameworks** and **/NextLibrary/Frameworks** (both prefaced by $NEXT_ROOT on Windows NT), plus any that you specify with **-F**.

-nostartfiles
Do not use the standard system startup files when linking. The standard system libraries are used normally, unless **-nostdlib** or **-nodefaultlibs** is used.

-nodefaultlibs
Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files are used normally, unless **-nostartfiles** is used.

-nostdlib Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker.

One of the standard libraries bypassed by **-nostdlib** and **-nodefaultlibs** is **libgcc.a**, a library of internal subroutines that GNU CC uses to overcome shortcomings of particular machines, or special needs for some languages. In most cases, you need **libgcc.a** even when you want to avoid other standard libraries. In other words, when you specify **-nostdlib** or **-nodefaultlibs** you should usually specify **-lgcc** as well. This ensures that you have no unresolved references to internal GNU CC library subroutines. (For example, **__main**, used to ensure C++ constructors will be called)

-s Remove all symbol table and relocation information from the executable.

-static On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.

-shared Produce a shared object which can then be linked with other objects to form an executable. Only a few systems support this option.

-symbolic Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option **-Xlinker -z -Xlinker defs**). Only a few systems support this option.

-undefined error, **-undefined warning**, **-undefined suppress**

Controls the behavior of the linker when symbols are undefined and cannot be resolved. **-undefined error** stipulates the default behavior, which causes the linker to generate an error message; no executable is produced. **-undefined warning** causes an executable to be generated, along with a warning indicating the unresolved symbols. **-undefined suppress** causes the executable to be generated, with no warning about unresolved symbols.

On Windows NT, **-undefined warning** and **-undefined suppress** are synonymous.

-Xlinker *option*

Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GNU CC does not know how to recognize.

If you want to pass an option that takes an argument, you must use **-Xlinker** twice, once for the option and once for the argument. For example, to pass **-assert definitions**, you must write **-Xlinker -assert -Xlinker definitions**. It does not work to write **-Xlinker "-assert definitions"**, because this passes the entire string as a single argument, which is not what the linker expects.

-Wl,*option*

Pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas.

-u *symbol*

Pretend the symbol *symbol* is undefined, to force linking of library modules to define it. You can use **-u** multiple times with different symbols to force loading of additional library modules.

Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

-I*directory*

Add the directory *directory* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one **-I** option, the directories are scanned in left-to-right order; the standard system directories come after.

When compiling a C++ file (extension **.C**, **.M**, or **.cc**), the compiler adds **NextDeveloper/Headers/g++** to its header search path. This allows **libg++** classes to be used without having to specify additional command-line options.

-I- Any directories you specify with **-I** options before the **-I-** option are searched only for the case of **#include "***file***"**; they are not searched for **#include** <*file*>.

If additional directories are specified with **-I** options after the **-I-**, these directories are searched for all **#include** directives. (Ordinarily *all* **-I** directories are used this way.)

In addition, the **-I-** option inhibits the use of the current directory (where the current input file came from) as the first search directory for **#include "***file***"**. There is no way to override this effect of **-I-**. With **-I**. you can specify searching the directory which was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

-I- does not inhibit the use of the standard system directories for header files. Thus, **-I-** and **-nostdinc** are independent.

-L*dir* Add directory *dir* to the list of directories to be searched for **-l**.

-F*dir* Add the directory *dir* to the head of the list of directories to be searched for frameworks. If you use more than one **-F** option, the directories are scanned in left-to-right order; the standard framework directories (**LocalLibrary/Frameworks**, followed by **NextLibrary/Frameworks**) come after.

In your Objective-C code, include framework headers using the following format:

```
#include <framework/include_file.h>
```

Where *framework* is the name of the framework (such as “AppKit” or “Foundation”—don’t include the extension) and *include_file* is the name of the file to be included.

-B*prefix*

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms **cpp**, **cc1**, **as** and **ld**. It tries *prefix* as a prefix for each program it tries to run, both with and without *machine*/*version*/ (see “Specifying Target Machine and Compiler Version”).

For each subprogram to be run, the compiler driver first tries the **-B** prefix, if any. If that name is not found, or if **-B** was not specified, the driver tries two standard prefixes, which are **/usr/lib/gcc/** and **/usr/local/lib/gcc-lib/**. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your **PATH** environment variable.

-B prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into **-L** options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into **-isystem** options for the preprocessor. In this case, the compiler appends **include** to the prefix.

The run-time support file **libgcc.a** can also be searched for using the **-B** prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means.

Another way to specify a prefix much like the **-B** prefix is to use the environment variable **GCC_EXEC_PREFIX**. See “Environment Variables Affecting GNU CC”.

Hardware Models and Configurations

On OPENSTEP for Mach, you specify the target architecture you are compiling for with **-arch** *arch_type*. The list of acceptable values for *arch_type* includes anything that **arch** can return (see **arch(3)** for more information), Typically, *arch_type* would be either **m68k, i386** (**i386** represents the processor family which includes the **i486** and **Pentium** processors), or **sparc**.

The option **-arch** *arch_type* specifies the target architecture, *arch_type*, of the operations to be performed. The operations affected by **-arch** are: preprocessing, precompiling, compiling, assembling, and linking. The specification of multiple architectures results in the production of “fat” output files and the creation of multiple “thin'” intermediate files from each stage. It is an error to use **-E**, **-S**, **-M**, and **-MM** with multiple architectures as the output form is textual in these cases.

In addition, each of these target machine types can have its own special options, starting with **-m**, to choose among various hardware models or configurations—for example, 68010 vs 68020, floating coprocessor or none. A single installed version of the compiler can compile for any model or configuration, according to the options specified.

Some configurations of the compiler also support additional special options, usually for compatibility with other compilers on the same platform.

These options are defined by the macro **TARGET_SWITCHES** in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

M680x0 Options

These are the -m options defined for the 68000 series. The default values for these options depends on which style of 68000 was selected when the compiler was configured; the defaults for the most common choices are given below.

-m68881 Generate output containing 68881 instructions for floating point. This is the default for most 68020 systems unless **-nfp** was specified when the compiler was configured.

-m68030 Generate output for a 68030. This is the default when the compiler is configured for 68030-based systems.

-m68040 Generate output for a 68040. This is the default when the compiler is configured for 68040-based systems.

This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. If your 68040 does not have code to emulate those instructions, use **-m68040**.

-m68020-40
Generate output for a 68040, without using any of the new instructions. This results in code which can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68040.

-msoft-float
Generate output containing library calls for floating point.

-mshort Consider type **int** to be 16 bits wide, like **short int**.

-mnobitfield
Do not use the bit-field instructions.

-mbitfield Do use the bit-field instructions. This is the default if you use a configuration designed for a 68020.

-mrtd Use a different function-calling convention, in which functions that take a fixed number of arguments return with the **rtd** instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including **printf**); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

SPARC Options

These -m switches are supported on the SPARC:

-mno-app-regs -mapp-regs
Specify **-mapp-regs** to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. This is the default.

To be fully SVR4 ABI compliant at the cost of some performance loss, specify **-mno-app-regs**. You should compile libraries and system software with this option.

-mfpu -mhard-float
Generate output containing floating point instructions. This is the default.

-mno-fpu -msoft-float
Generate output containing library calls for floating point.

Warning: The requisite libraries are not available for all SPARC targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets **sparc-*-aout** and **sparclite-*-*** do provide software floating point support.

-msoft-float changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile **libgcc.a**, the library that comes with GNU CC, with **-msoft-float** in order for this to work.

-mhard-quad-float

Generate output containing quad-word (long double) floating point instructions.

-msoft-quad-float

Generate output containing library calls for quad-word (long double) floating point instructions. The functions called are those specified in the SPARC ABI. This is the default.

As of this writing, there are no sparc implementations that have hardware support for the quad-word floating point instructions. They all invoke a trap handler for one of these instructions, and then the trap handler emulates the effect of the instruction. Because of the trap handler overhead, this is much slower than calling the ABI library routines. Thus the **-msoft-quad-float** option is the default.

-mno-epilogue -mepilogue

With **-mepilogue** (the default), the compiler always emits code for function exit at the end of each function. Any function exit in the middle of the function (such as a return statement in C) will generate a jump to the exit code at the end of the function.

With **-mno-epilogue**, the compiler tries to emit exit code inline at every function exit.

-mno-flat -mflat

With **-mflat**, the compiler does not generate save/restore instructions and will use a "flat" or single register window calling convention. This model uses %i7 as the frame pointer and is compatible with the normal register window model. Code from either may be intermixed although debugger support is still incomplete. The local registers and the input registers (0-5) are still treated as "call saved" registers and will be saved on the stack as necessary.

With **-mno-flat** (the default), the compiler emits save/restore instructions (except for leaf functions) and is the normal mode of operation.

-mno-unaligned-doubles -munaligned-doubles
Assume that doubles have 8 byte alignment. This is the default.

With **-munaligned-doubles**, GNU CC assumes that doubles have 8 byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4 byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers. It is not the default because it results in a performance loss, especially for floating point code.

-mv8 -msparclite
These two options select variations on the SPARC architecture.

By default (unless specifically configured for the Fujitsu SPARClite), GCC generates code for the v7 variant of the SPARC architecture.

-mv8 will give you SPARC v8 code. The only difference from v7 code is that the compiler emits the integer multiply and integer divide instructions which exist in SPARC v8 but not in SPARC v7.

-msparclite will give you SPARClite code. This adds the integer multiply, integer divide step and scan (**ffs**) instructions which exist in SPARClite but not in SPARC v7.

-mcypress -msupersparc
These two options select the processor for which the code is optimised.

With **-mcypress** (the default), the compiler optimizes code for the Cypress CY7C602 chip, as used in the SparcStation/SparcServer 3xx series. This is also appropriate for the older SparcStation 1, 2, IPX etc.

With **-msupersparc** the compiler optimizes code for the SuperSparc cpu, as used in the SparcStation 10, 1000 and 2000 series. This flag also enables use of the full SPARC v8 instruction set.

In a future version of GCC, these options will very likely be renamed to **-mcpu=cypress** and **-mcpu=supersparc**.

These **-m** switches are supported in addition to the above on SPARC V9 processors:

-mmedlow
Generate code for the Medium/Low code model: assume a 32 bit address space. Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

-mmedany
Generate code for the Medium/Anywhere code model: assume a 32 bit text segment starting at offset 0, and a 32 bit data segment starting anywhere (determined at link time). Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

It is very likely that a future version of GCC will rename this option.

-mint64 Types long and int are 64 bits.

-mlong32 Types long and int are 32 bits.

-mlong64 -mint32
Type long is 64 bits, and type int is 32 bits.

-mstack-bias -mno-stack-bias
With **-mstack-bias**, GNU CC assumes that the stack pointer, and frame pointer if present, are offset by -2047 which must be added back when making stack frame references. Otherwise, assume no such offset is present.

Intel 386 Options

These -m options are defined for the i386 family of computers:

-m486 -m386
Control whether or not code is optimized for a 486 instead of an 386. Code generated for a 486 will run on a 386 and vice versa.

-mieee-fp -mno-ieee-fp
Control whether or not the compiler uses IEEE floating point comparisons. These handle correctly the case where the result of a comparison is unordered.

-mno-fp-ret-in-387
Do not use the FPU registers for return values of functions.

The usual calling convention has functions return values of types **float** and **double** in an FPU register, even if there is no FPU. The idea is that the operating system should emulate an FPU.

The option **-mno-fp-ret-in-387** causes such values to be returned in ordinary CPU registers instead.

-mno-fancy-math-387
Some 387 emulators do not support the **sin**, **cos** and **sqrt** instructions for the 387. Specify this option to avoid generating those instructions. This option is the default on FreeBSD. As of revision 2.6.1, these instructions are not generated unless you also use the **-ffast-math** switch.

-malign-double -mno-align-double
Control whether GNU CC aligns **double**, **long double**, and **long long** variables on a two word boundary or a one word boundary. Aligning **double** variables on a two word boundary will produce code that runs somewhat faster on a Pentium at the expense of more memory.

Warning: If you use the **-malign-double** switch, structures containing the above types will be aligned differently than the published application binary interface specifications for the 386.

-munaligned-text
(Not available on Windows NT) Turns off all alignment for instructions. Occasionally this may be interesting if the code size is significant in low-level stuff.

-msvr3-shlib -mno-svr3-shlib
Control whether GNU CC places uninitialized locals into **bss** or **data**. **-msvr3-shlib** places these locals into **bss**. These options are meaningful only on System V Release 3.

-mno-wide-multiply -mwide-multiply
Control whether GNU CC uses the **mul** and **imul** that produce 64 bit results in **eax:edx** from 32 bit operands to do **long long** multiplies and 32-bit division by constants.

-mrtd Use a different function-calling convention, in which functions that take a fixed number of arguments return with the **ret** NUM instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

You can specify that an individual function is called with this calling sequence with the function attribute **stdcall**. You can also override the **-mrtd** option by using the function attribute **cdecl**. See “Declaring Attributes of Functions”

Warning: This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including **printf**); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

-mreg-alloc=*regs*

Control the default allocation order of integer registers. The string *regs* is a series of letters specifying a register. The supported letters are: **a** allocate EAX; **b** allocate EBX; **c** allocate ECX; **d** allocate EDX; **S** allocate ESI; **D** allocate EDI; **B** allocate EBP.

-mregparm=*num*

Control how many registers are used to pass integer arguments. By default, no registers are used to pass arguments, and at most 3 registers can be used. You can control this behavior for a specific function by using the function attribute regparm. See “Declaring Attributes of Functions”

Warning: If you use this switch, and *num* is nonzero, then you must build all modules with the same value, including any libraries. This includes the system libraries and startup modules.

-malign-loops=*num*

Align loops to a 2 raised to a *num* byte boundary. If **-malign-loops** is not specified, the default is 2.

-malign-jumps=*num*

Align instructions that are only jumped to to a 2 raised to a *num* byte boundary. If **-malign-jumps** is not specified, the default is 2 if optimizing for a 386, and 4 if optimizing for a 486.

-malign-functions=*num*

Align the start of functions to a 2 raised to *num* byte boundary. If **-malign-jumps** is not specified, the default is 2 if optimizing for a 386, and 4 if optimizing for a 486.

HPPA Options

These **-m** options are defined for the HPPA family of computers:

-mpa-risc-1-0
: Generate code for a PA 1.0 processor.

-mpa-risc-1-1
: Generate code for a PA 1.1 processor.

-mjump-in-delay
: Fill delay slots of function calls with unconditional jump instructions by modifying the return pointer for the function call to be the target of the conditional jump.

-mmillicode-long-calls
: Generate code which assumes millicode routines can not be reached by the standard millicode call sequence, linker-generated long-calls, or linker-modified millicode calls. In practice this should only be needed for dynamicly linked executables with extremely large SHLIB_INFO sections.

-mdisable-fpregs
: Prevent floating point registers from being used in any manner. This is necessary for compiling kernels which perform lazy context switching of floating point registers. If you use this option and attempt to perform floating point operations, the compiler will abort.

-mdisable-indexing
: Prevent the compiler from using indexing address modes. This avoids some rather obscure problems when compiling MIG generated code under MACH.

-mfast-indirect-calls
: Generate code which performs faster indirect calls. Such code is suitable for kernels and for static linking. The fast indirect call code will fail miserably if it’s part of a dynamically linked executable and in the presense of nested functions.

-mportable-runtime
: Use the portable calling conventions proposed by HP for ELF systems.

-mgas Enable the use of assembler directives only GAS understands.

-mschedule=*cpu type*

Schedule code according to the constraints for the machine type *cpu type*. The choices for *cpu type* are **700** for 7N0 machines, **7100** for 7N5 machines, and **7100** for 7N2 machines. **700** is the default for *cpu type*.

Note the **7100LC** scheduling information is incomplete and using **7100LC** often leads to bad schedules. For now it's probably best to use **7100** instead of **7100LC** for the 7N2 machines.

-msoft-float

Generate output containing library calls for floating point.

Warning: The requisite libraries are not available for all HPPA targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded target **hppa1.1-*-pro** does provide software floating point support.

-msoft-float changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile **libgcc.a**, the library that comes with GNU CC, with **-msoft-float** in order for this to work.

Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of **-ffoo** would be **-fno-foo**. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing **no-** or adding it.

-dynamic, -static

The compiler generates position-independent code by default when it builds libraries, bundles, and executables. You can control the code generation style using the **-dynamic** and **-static** compiler flags; **-dyamic** specifies that position-independent code generation is to be used, whereas **-static** specifies position-dependent code generation.

If you are building drivers and kernel servers, be sure to include **-static** on the command line so that position-dependent code is generated. Compilation with the **-dynamic** option assumes that the dynamic link editor (**/usr/lib/dyld**) is present in the running program, and that is not the case for modules to be loaded into the kernel.

-fpcc-struct-return

Return "short" **struct** and **union** values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between GNU CC-compiled files and files compiled with other compilers.

The precise convention for returning structures in memory depends on the target configuration macros.

Short structures and unions are those whose size and alignment match that of some integer type.

-freg-struct-return

Use the convention that **struct** and **union** values are returned in registers when possible. This is more efficient for small structures than **-fpcc-struct-return**.

If you specify neither **-fpcc-struct-return** nor its contrary **-freg-struct-return**, GNU CC defaults to whichever convention is standard for the target. If there is no standard convention, GNU CC defaults to **-fpcc-struct-return**, except on targets where GNU CC is the principal compiler. In those cases, it choses the more efficient register return alternative.

-fshort-enums

Allocate to an **enum** type only as many bytes as it needs for the declared range of possible values. Specifically, the **enum** type will be equivalent to the smallest integer type which has enough room.

-fshort-double

Use the same size for **double** as for **float**.

-fshared-data

Requests that the data and non-**const** variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

-fno-common

Allocate even uninitialized global variables in the bss section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without **extern**) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

-fno-ident

Ignore the **#ident** directive.

-fno-gnu-linker

Do not output global initializations (such as C++ constructors and destructors) in the form used by the GNU linker (on systems where the GNU linker is the standard method of handling them). Use this option when you want to use a non-GNU linker, which also requires using the **collect2** program to make sure the system linker includes constructors and destructors. (**collect2** is included in the GNU CC distribution.) For systems which *must* use **collect2**, the compiler driver **gcc** is configured to do this automatically.

-finhibit-size-directive

Don't output a **.size** assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling **crtstuff.c**; you should not need to use it for anything else.

-fverbose-asm

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

-fvolatile

Consider all memory references through pointers to be volatile.

-fvolatile-global

Consider all memory references to extern and global data items to be volatile.

-fpic Generate position-independent code (PIC) suitable for use in a shared library. Such code accesses all constant addresses through a global offset table (GOT). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that **-fpic** does not work; in that case, recompile with **-fPIC** instead. (These maximums are 8k on the Sparc and 32k on the m68k. The 386 has no such limit.)

-fPIC Emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table.

-ffixed-*reg*

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the **REGISTER_NAMES** macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

-fcall-used-*reg*

Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

-fcall-saved-*reg*

Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

-fpack-struct

Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code suboptimal, and the offsets of structure members won't agree with system libraries.

+e0 +e1

Control whether virtual function definitions in classes are used to generate code, or only to define interfaces for their callers. (C++ only).

These options are provided for compatibility with **cfront** 1.x usage; the recommended alternative GNU C++ usage is in flux. See "Declarations and Definitions in One Header".

With **+e0**, virtual function definitions in classes are declared **extern**; the declaration is used only as an interface specification, not to generate code for the virtual functions (in this compilation).

With **+e1**, G++ actually generates the code implementing virtual functions defined in the code, and makes them publicly visible.

Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as **-B**, **-I** and **-L** (see "Options for Directory Search"). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GNU CC.

TMPDIR If TMPDIR is set, it specifies the directory to use for temporary files. GNU CC uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

GCC_EXEC_PREFIX

If **GCC_EXEC_PREFIX** is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

If GNU CC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

The default value of **GCC_EXEC_PREFIX** is *prefix***lib/gcc-lib/** where *prefix* is the value of **prefix** when you ran the **configure** script.

Other prefixes specified with **-B** take precedence over this prefix.

This prefix is also used for finding files such as **crt0.o** that are used for linking.

In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with **/usr/local/lib/gcc-lib** (more precisely, with the value of **GCC_INCLUDE_DIR**), GNU CC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with **-Bfoo/**, GNU CC will search **foo/bar** where it would normally search **/usr/local/lib/bar**. These alternate directories are searched first; the standard directories come next.

COMPILER_PATH

The value of **COMPILER_PATH** is a colon-separated list of directories, much like **PATH**. GNU CC tries the directories thus specified when searching for subprograms, if it can't find the subprograms using **GCC_EXEC_PREFIX**.

LIBRARY_PATH

The value of **LIBRARY_PATH** is a colon-separated list of directories, much like **PATH**. When configured as a native compiler, GNU CC tries the directories thus specified when searching for special linker files, if it can't find them using **GCC_EXEC_PREFIX**. Linking using GNU CC also uses these directories when searching for ordinary libraries for the **-l** option (but directories specified with **-L** come first).

C_INCLUDE_PATH CPLUS_INCLUDE_PATH OBJC_INCLUDE_PATH

These environment variables pertain to particular languages. Each variable's value is a colon-separated list of directories, much like **PATH**. When GNU CC searches for header files, it tries the directories listed in the variable for the language you are using, after the directories specified with **-I** but before the standard header file directories.

DEPENDENCIES_OUTPUT

If this variable is set, its value specifies how to output dependencies for Make based on the header files processed by the compiler. This output looks much like the output from the **-M** option (see "Options Controlling the Preprocessor"), but it goes to a separate file, and is in addition to the usual results of compilation.

The value of **DEPENDENCIES_OUTPUT** can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form *file target*, in which case the rules are written to file *file* using *target* as the target name.

C Programming Notes

This section contains miscellaneous notes about programming in C with NeXT's version of the GNU C compiler. It also describes some incompatibilities between GNU C and traditional non-ANSI versions of C.

String Constants and Static Strings

GNU CC normally makes string constants read-only, and if several identical string constants are used, GNU CC stores only one copy of the string.

Some C libraries incorrectly write into string constants. The best solution to this problem is to use character array variables with initialization strings instead of string constants. If this isn't possible, use the **-fwritable-strings** flag, which directs GNU CC to handle string constants the way most C compilers do.

Also note that initialized strings are normally put in the text segment by the GNU compiler, and attempts to write to them cause segmentation faults. If your program depends on being able to write initialized strings, there are two ways to get around this problem:

- Compile your program with the **-fwritable-strings** compiler option.
- Declare your string as an unbounded array of **char**s, which will force it to appear in the data segment:

```
char *non_writable = "You can't write this string";
char writable[] = "You can write this string";
```

Function Prototyping

Function prototypes are a new and important feature of the ANSI standard. You should use function prototypes in your C programs, so the compiler can generate more efficient code (because it knows what the called function is expecting). The compiler can also warn you when you pass the wrong number or wrong type of arguments to a function.

Extra care must be taken in using function prototypes. Be sure to follow these rules:

- Each function must be declared explicitly (with a prototype) before calling the function. Multiple declarations must agree exactly. Incorrect code can be generated by a call that isn't prototyped if the function itself is declared as a prototype.
- The parameter declarations for the prototyped function must be in the same form as the prototype declaration.

Here are a few points about prototyping that might cause you some trouble.

- You might think it's a bug when GNU CC reports an error for code like this:

```
int foo (short);

int foo (x)
    short x;
{ . . . }
```

  The error message is correct. The code is wrong because the old-style nonprototype definition passes subword integers in their promoted types. In other words, the argument is really an **int**, not a **short**. The correct prototype is this:

```
int foo (int)
```

- You might think it's a bug when GNU CC reports an error for code like this:

```
int foo (struct mumble *);

struct mumble { . . . };

int foo (struct mumble *x);
{ . . . }
```

  This code is also wrong. Because of the scope of **struct mumble**, the prototype is limited to the argument list containing it. It doesn't refer to the **struct mumble** defined with file scope immediately below—they are two unrelated types with similar names in different scopes. But in the definition of **foo**, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype don't match and you get

an error. You can make the code work by simply moving the definition of **struct mumble** above the prototype.

"Suggested Reading" lists several C books that provide detailed information about the use (and abuse) of function prototypes.

Automatic Register Allocation

When you use **setjmp()** and **longjmp()**, the only automatic variables guaranteed to remain valid are those declared **volatile**. This is a consequence of automatic register allocation. If you use the **-W** option with the **-O** option, you'll get a warning when GNU CC thinks such a problem is possible. For example:

```
jmp_buf j;

foo ()
{
    int a, b;

    a = fun1 ();
    if (setjmp (j))
        return a;

    a = fun2 ();
    /* longjmp (j) may occur in fun3. */
    return a + fun3 ();
}
```

Here, **a** may or may not be restored to its first value when the **longjmp()** function is called. If **a** is allocated in a register, its first value is restored; otherwise, it keeps the last value stored in it.

Declarations of External Variables and Functions

Declarations of external variables and functions within a block apply only to the block containing the declaration (in some C compilers, such declarations affect the whole file). ANSI C states that external declarations should obey normal scoping rules. For example:

```
{
    {
        extern int a;
        a = 0;
    }
    a = 1;     /* Illegal */
}
```

You can use the **-traditional** option if you want all **extern** declarations to be treated as global.

typedef and Type Modifiers

In traditional C, you can combine **unsigned**, for example, with a **typedef** name as shown here:

```
typedef long int Int32;
unsigned Int32 i;      /* Illegal in ANSI C*/
```

In ANSI C this isn't allowed: **unsigned** and other type modifiers require an explicit **int**. Because this criterion is expressed by Bison grammar rules rather than C code, the **-traditional** flag can't alter it.

The same difficulty applies to **typedef** names used as function parameters.

Identifying the Compiler Version

The compiler has additional predefined macros that can be used to determine the release version of the compiler (these macros are not available on Windows NT). *Every effort should be made to minimize the use of these macros.* For each release of the compiler there will be a macro defined such as **NX_COMPILER_RELEASE_3_0** and **NX_COMPILER_RELEASE_3_1**. There will also be a macro **NX_CURRENT_COMPILER_RELEASE**. One can conditionally compile code by numerically comparing these macros. For example:

```
#if NX_CURRENT_COMPILER_RELEASE > NX_COMPILER_RELEASE_3_0
    ...
#endif
```

Writing Architecture-Independent Code

This compiler predefines new macros to aid in writing architecture-independent code.

__ARCHITECTURE__
: In addition to the existing predefines which identify specific target architectures (for example, **m68k**, **i386**), the compiler also predefines the macro **__ARCHITECTURE__** to be a string constant identifying the target architecture (“m68k”, “i386”). This macro is used by system header files to include the architecture-specific files without having to enumerate all supported architectures.

__BIG_ENDIAN__, **__LITTLE_ENDIAN__**
: The compiler predefines either **__BIG_ENDIAN__** or **__LITTLE_ENDIAN__**, as appropriate for the target architecture.

Objective-C Programming Notes

Accessing Instance Variables in Class Methods

It used to be common programming style in Objective-C to assign to **self** in a class method and then access instance variables. This is bad style because **self** in the context of a class method stands for the class object—and shouldn't be redefined to stand for a particular instance of the object.

Here is an example of this *bad style*:

```
@implementation Oval : Object {
    int x;
}

+new {
    self = [super new];// Now self refers to a class instance
    ...
    x = 4;// Assigns an instance variable
} ...
@end
...
```

```
x = [Oval new];// Create an Oval object
```

To discourage this anachronistic use, the compiler issues a warning if an instance variable is referenced in a class method.

Here's a better way to instantiate an object:

```
x = [[Oval alloc] init];
```

See *Object-Oriented Programming and the Objective C Language* for more details.

Syntax Checking

The Objective-C compiler's syntax checking disallows the nesting of **@interface** and **@implementation** blocks.

Sending Objective-C Messages to Converted C++ Objects

You can send an Objective-C message to a C++ object that has been converted by a conversion operator ("a smart pointer"). In the following example, the C++ **ptrSquare** object **aSquare** is implicitly converted to the Objective-C type **Square*** using the conversion operator **Square*()**. The converted object receives the message **calculateArea**:

```
@interface Square {
    id a;
}
...
@end

class ptrSquare {
    Square* value;
public:
    operator Square*();
};

square (ptrSquare aSquare) {
    float z = [aSquare calculateArea]; // invokes operator Square*()
}
```

Due to the conversion, the compiler acts as if **aSquare** is statically typed to **Square*** in the message expression.

The above example uses only one conversion operator: **operator Square***. You should avoid having multiple conversion operators in the same class that produce different pointer types—the compiler may choose the wrong conversion operator and not produce the desired type. If you need more than one conversion type, you must use an **operator id** conversion operator—the compiler chooses this over an operator converting to any other Objective-C class pointer type. If the class **ptrSquare** implemented other operator *x***()** conversions besides **operator Square*()**, it would also have to implement an **operator id** conversion so the compiler would know which conversion to look for.

Conversion operators allow you to implement so called "smart pointers" to Objective-C objects. Smart pointers are objects that act like pointers and perform some other action in addition whenever an object is accessed through them. For more information on smart pointers, see Bjarne Stroustrup's *The C++ Programming Language, Second Edition* (Addison-Wesley, 1991).

Extensions to the C Language Family

GNU C provides several language features not found in ANSI standard C. (The **-pedantic** option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro **__GNUC__**, which is always defined under GNU CC.

These extensions are available in C and Objective-C. Most of them are also available in C++. See "Extensions to the C++ Language" for extensions that apply *only* to C++.

Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z; if (y > 0) z = y; else z = - y; z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of **foo ()**.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type **void**, and thus effectively no value.)

This feature is especially useful in making macro definitions “safe” (so that they evaluate each operand exactly once). For example, the “maximum” function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either A or B twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let’s assume **int**), you can define the macro safely as follows:

```
#define maxint(a,b) \ ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don’t know the type of the operand, you can still do this, but you must use **typeof** (see “Referring to a Type with “typeof””) or type naming (see “Naming an Expression’s Type”).

Locally Declared Labels

Each statement expression is a scope in which “local labels” can be declared. A local label is simply an identifier; you can jump to it with an ordinary **goto** statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

```
__label__ LABEL;
```

or

```
__label__ LABEL1, LABEL2, ...;
```

Local label declarations must come at the beginning of the statement expression, right after the **({**‘, before any ordinary declarations.

The label declaration defines the label *name*, but does not define the label itself. You must do this in the usual way, with **LABEL:**, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a **goto** can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target) \
({ \
__label__ found; \
    typeof (target) _SEARCH_target = (target); \
    typeof (*(array)) *_SEARCH_array = (array); \
    int i, j; \
    int value; \
    for (i = 0; i < max; i++) \
       for (j = 0; j < max; j++) \
          if (_SEARCH_array[i][j] == _SEARCH_target) \
             { value = i; goto found; } \
    value = -1; \
found: \
    value; \
})
```

Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator **&&**. The value has type **void ***. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr; ... ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement(1), **goto *EXP;**. For example,

```
goto *ptr;
```

Any expression of type **void *** is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

Such an array of label values serves a purpose much like that of the **switch** statement. The **switch** statement is cleaner, so use that rather than an array unless the problem does not fit a **switch** statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You can use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

Note: The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

Nested Functions

A “nested function” is a function defined inside another function. (Nested functions are not supported for GNU C++.) The nested function’s name is local to the block where it is defined. For example, here we define a nested function named **square**, and call it twice:

```
foo (double a, double b) {
    double square (double z) {
       return z * z;
    }
    return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called “lexical scoping”. For example, here we show a nested function which uses an inherited variable named **offset**:

```
bar (int *array, int offset, int size) {
    int access (int *array, int index) {
       return array[index + offset];
    }
    int i;
    ...
    for (i = 0; i < size; i++)
```

```
        ...
    access (array, i)
    ...
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size) {
    void store (int index, int value) {
       array[index] = value;
    }
    intermediate (store, size);
}
```

Here, the function **intermediate** receives the address of **store** as an argument. If **intermediate** calls **store**, the arguments given to **store** are used to store into **array**. But this technique works only so long as the containing function (**hack**, in this example) does not exit.

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

GNU CC implements taking the address of a nested function using a technique called "trampolines". A paper describing them is available from **maya.idiap.ch** in directory **pub/tm**b, file **usenix88-lexic.ps.Z**.

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see "Locally Declared Labels"). Such a jump returns instantly to the containing function, exiting the nested function which did the **goto** and any intermediate functions as well. Here is an example:

```
bar (int *array, int offset, int size) {
__label__ failure;
    int access (int *array, int index) {
       if (index > size)
          goto failure;
       return array[index + offset];
    }
    int i;
```

```
    ...
    for (i = 0; i < size; i++)
       ...
    access (array, i)
    ...
    ...
    return 0; /* Control comes here from access */
              /* if it detects an error. */
failure:
    return -1;
}
```

A nested function always has internal linkage. Declaring one with **extern** is erroneous. If you need to declare the nested function before its definition, use **auto** (which is otherwise meaningless for function declarations).

```
bar (int *array, int offset, int size) {
__label__ failure;
    auto int access (int *, int);
    ...
    int access (int *array, int index) {
       if (index > size)
          goto failure;
       return array[index + offset];
    }
    ...
}
```

Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

__builtin_apply_args ()

This built-in function returns a pointer of type **void** * to data describing how to perform a call with the same arguments as were passed to the current function.

The function saves the arg pointer register, structure value address, and all registers that might be used to pass arguments to a function into a block of memory allocated on the stack. Then it returns the address of that block.

__builtin_apply (*function***,** *arguments***,** *size***)** This built-in function invokes *function* (type **void (*)()**) with a copy of the parameters described by *arguments* (type **void ***) and SIZE (type **int**).

The value of *arguments* should be the value returned by **__builtin_apply_args**. The argument *size* specifies the size of the stack argument data, in bytes.

This function returns a pointer of type **void *** to data describing how to return whatever value was returned by *function*. The data is saved in a block of memory allocated on the stack.

It is not always simple to compute the proper value for *size*. The value is used by **__builtin_apply** to compute the amount of data that should be pushed on the stack and copied from the incoming argument area.

__builtin_return (*result***)**
This built-in function returns the value described by *result* from the containing function. You should specify, for *result*, a value returned by **__builtin_apply**.

Naming an Expression's Type

You can give a name to the type of an expression using a **typedef** declaration with an initializer. Here is how to define NAME as a type name for the type of EXP:

```
typedef NAME = EXP;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
({typedef _ta = (a), _tb = (b); \
_ta _a = (a); _tb _b = (b); \
_a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for **a** and **b**.

Referring to a Type with “typeof”

Another way to refer to the type of an expression is with **typeof**. The syntax of using of this keyword looks like **sizeof**, but the construct acts semantically like a type name defined with **typedef**.

There are two ways of writing the argument to **typeof**: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that **x** is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to **int**.

If you are writing a header file that must work when included in ANSI C programs, write **__typeof__** instead of **typeof**. See “Alternate Keywords”.

A **typeof**-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of **sizeof** or **typeof**.

- This declares **y** with the type of what **x** points to.

  ```
  typeof (*x) y;
  ```

- This declares **y** as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares **y** as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

- It is equivalent to the following traditional C declaration:

  ```
  char *y[4];
  ```

- To see the meaning of the declaration using **typeof**, and why it might be a useful way to write, let’s rewrite it with these macros:

  ```
  #define pointer(T) typeof(T *) #define array(T, N) typeof(T [N])
  ```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, **array (pointer (char), 4)** is the type of arrays of 4 pointers to **char**.

Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

Standard C++ allows compound expressions and conditional expressions as lvalues, and permits casts to reference type, so use of this extension is deprecated for C++ code.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5 a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b) a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5 (a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if **a** has type **char ***, the following two expressions are equivalent:

```
(int)a = 5 (int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as **+=** applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5 (int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that **&(int)f** were permitted, where **f** has type **float**. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what **(int)f = 1** would do—that would convert 1 to floating point and store it. Rather than cause this inconsistency, we think it is better to prohibit use of **&** on a cast.

If you really do want an **int *** pointer with the address of **f**, you can simply write **(int *)&f**.

Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of **x** if that is nonzero; otherwise, the value of **y**.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Double-Word Integers

GNU C supports data types for integers that are twice as long as **long int**. Simply write **long long int** for a signed integer, or **unsigned long long int** for an unsigned integer. To make an integer constant of type **long long int**, add the suffix **LL** to the integer. To make an integer constant of type **unsigned long long int**, add the suffix **ULL** to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports fullword-to-doubleword a widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GNU CC.

There may be pitfalls when you use **long long** types for function arguments, unless you declare function prototypes. If a function expects type **int** for its argument, and you pass a value of type **long long int**, confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects **long long int** and you pass **int**. The best way to avoid such problems is to use prototypes.

Complex Numbers

GNU C supports complex data types. You can declare both complex integer types and complex floating types, using the keyword **__complex__**.

For example, **__complex__ double x;** declares **x** as a variable whose real part and imaginary part are both of type **double**. **__complex__ short int y;** declares **y** to have real and imaginary parts of type **short int**; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix **i** or **j** (either one; they are equivalent). For example, **2.5fi** has type **__complex__ float** and **3i** has type **__complex__ int**. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression *exp*, write **__real__** *exp*. Likewise, use **__imag__** to extract the imaginary part.

The operator **~** performs complex conjugation when used on a value with a complex type.

GNU CC can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). None of the supported debugging info formats has a way to represent noncontiguous allocation like this, so GNU CC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable's actual name is **foo**, the two fictitious variables are named **foo$real** and **foo$imag**. You can examine and set these two fictitious variables with your debugger.

A future version of GDB will know how to recognize such pairs and treat them as a single variable with a complex type.

Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
    int length;
    char contents[0];
};
{
    struct line *thisline =
       (struct line *) malloc (sizeof (struct line) + this_length);
    thisline->length = this_length;
}
```

In standard C, you would have to give **contents** a length of 1, which means either you waste space or complicate the argument to **malloc**.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE * concat_fopen (char *s1, char *s2, char *mode) {
    char str[strlen (s1) + strlen (s2) + 1];
    strcpy (str, s1);
    strcat (str, s2);
    return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function **alloca** to get an effect much like variable-length arrays. The function **alloca** is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with **alloca** exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and **alloca** in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with **alloca**.)

You can also use variable-length arrays as arguments to functions:

```
struct entry tester (int len, char data[len][len]) { ... }
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with **sizeof**.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list—another GNU extension.

```
struct entry tester (int len; char data[len][len], int len) { ... }
```

The **int len** before the semicolon is a "parameter forward declaration", and it serves the purpose of making the name **len** known when the declaration of **data** is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.

Macros with Variable Numbers of Arguments

In GNU C, a macro can accept a variable number of arguments, much as a function can. The syntax for defining the macro looks much like that used for a function. Here is an example:

```
#define eprintf(format, args...) \ fprintf (stderr, format , ## args)
```

Here **args** is a "rest argument": it takes in zero or more arguments, as many as the call contains. All of them plus the commas between them form the value of **args**, which is substituted into the macro body where **args** is used. Thus, we have this expansion:

```
eprintf ("%s:%d: ", input_file_name, line_number) ==> fprintf
(stderr, "%s:%d: " , input_file_name, line_number)
```

Note that the comma after the string constant comes from the definition of **eprintf**, whereas the last comma comes from the value of **args**.

The reason for using ## is to handle the case when **args** matches no arguments at all. In this case, **args** has an empty value. In this case, the second comma in the definition becomes an embarrassment: if it got through to the expansion of the macro, we would get something like this:

```
fprintf (stderr, “success!\n” , )
```

which is invalid C syntax. ## gets rid of the comma, so we get the following instead:

```
fprintf (stderr, “success!\n”)
```

This is a special feature of the GNU C preprocessor: ## before a rest argument that is empty discards the preceding sequence of non-whitespace characters from the macro definition. (If another macro argument precedes, none of it is discarded.)

It might be better to discard the last preprocessor token instead of the last preceding sequence of non-whitespace characters. We advise you to write the macro definition so that the preceding sequence of non-whitespace characters is just a single token, so that the meaning will not change if GNU changes the definition of this feature.

Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary **&** operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {
    int a[4];
};

struct foo f();

bar (int index) {
    return f().a[index];
}
```

Arithmetic on “void”- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to **void** and on pointers to functions. This is done by treating the size of a **void** or of a function as 1.

A consequence of this is that **sizeof** is also allowed on **void** and on function types, and returns 1.

The option **-Wpointer-arith** requests a warning if these extensions are used.

Non-Constant Initializers

As in standard C++, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g) {
    float beat_freqs[2] = { f-g, f+g };
    ...
}
```

Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that **struct foo** and **structure** are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a **struct foo** with a constructor:

```
structure = ((struct foo) {x + y, ‘a, 0});
```

This is equivalent to writing the following:

```
{ struct foo temp = {x + y, ‘a, 0}; structure = temp; }
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { “x”, “y”, “z” };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a **switch** statement,

while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. In GNU C you can give the elements in any order, specifying the array indices or structure field names they apply to.

Note: This extension is not implemented in GNU C++, Objective-C, or Objective-C++.

To specify an array index, write **[***index***]** or **[***index***] =** before the element value. For example,

```
int a[6] = { [4] 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

To initialize a range of elements to the same value, write **[***first* **...** *last***] =** *value*. For example,

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with *fieldname*: before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { y: yvalue, x: xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

Another syntax which has the same meaning is **.***fieldname* **=**., as shown here:

```
struct point p = { .y = yvalue, .x = xvalue };
```

You can also use an element label (with either the colon syntax or the period-equal syntax) when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };
union foo f = { d: 4 };
```

will convert 4 to a **double** to store it in the union using the second element. By contrast, casting 4 to type **union foo** would store it into the union as the integer **i**, since it is an integer. (See “Cast to a Union Type”.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a label applies to the next consecutive element of the array or structure. For example,

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an **enum** type. For example:

```
int whitespace[256] = {
    [' '] = 1,
    ['\t'] = 1,
    ['\h'] = 1,
    ['\f'] = 1,
    ['\n'] = 1,
    ['\r'] = 1
};
```

Case Ranges

You can specify a range of consecutive values in a single **case** label, like this:

```
case LOW ... HIGH:
```

This has the same effect as the proper number of individual **case** labels, one for each integer value from LOW to HIGH, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case ‘A ... ‘Z’:
```

Be careful: Write spaces around the **...**, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with **union** *tag* or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See “Constructor Expressions”.)

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both **x** and **y** can be cast to type **union** foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x == u.i = x u = (union foo) y == u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword **__attribute__** allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. Eight attributes, **noreturn**, **const**, **format**, **section**, **constructor**, **destructor**, **unused** and **weak** are currently defined for functions. Other attributes, including **section** are supported for variables declarations (see "Specifying Attributes of Variables") and for types (see "Specifying Attributes of Types").

You may also specify attributes with **__** preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use **__noreturn__** instead of **noreturn**.

noreturn A few standard library functions, such as **abort** and **exit**, cannot return. GNU CC knows this automatically. Some programs define their own functions that never return. You can declare them **noreturn** to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));
void fatal (...) {
    ...
    /* Print error message. */
    ...
    exit (1);
}
```

The **noreturn** keyword tells the compiler to assume that **fatal** cannot return. It can then optimize without regard to what would happen if **fatal** ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

Do not assume that registers saved by the calling function are restored before calling the **noreturn** function.

It does not make sense for a **noreturn** function to have a return type other than **void**.

The attribute **noreturn** is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```
typedef void voidfn ();
```

```
volatile voidfn fatal;
```

const Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute **const**. For example,

```
int square (int) __attribute__ ((const));
```

says that the hypothetical function **square** is safe to call fewer times than the program says.

The attribute **const** is not implemented in GNU C versions earlier than 2.5. An alternative way to declare that a function has no side effects, which works in the current version and in some older versions, is as follows:

```
typedef int intfn ();
extern const intfn square;
```

This approach does not work in GNU C++ from 2.6.0 on, since the language specifies that the **const** must be attached to the return value.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared **const**. Likewise, a function that calls a non-**const** function usually must not be **const**. It does not make sense for a **const** function to return **void**.

format (*archetype***,** *string-index***,** *first-to-check***)**

The **format** attribute specifies that a function takes **printf** or **scanf** style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int my_printf (void *my_object, const char
*my_format, ...) __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to **my_printf** for consistency with the **printf** style format string argument **my_format**.

The parameter *archetype* determines how the format string is interpreted, and should be either **printf** or **scanf**. The parameter *string-index* specifies which argument is the format string argument (starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as **vprintf**), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (**my_format**) is the second argument of the function **my_print**, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The **format** attribute allows you to identify your own functions which take format strings as arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions **printf**, **fprintf**, **sprintf**, **scanf**, **fscanf**, **sscanf**, **vprintf**, **vfprintf** and **vsprintf** whenever such warnings are requested (using **-Wformat**), so there is no need to modify the header file **stdio.h**.

section ("section-name")

Normally, the compiler places the code it generates in the **text** section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The **section** attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function **foobar** in the **bar** section.

Some file formats do not support arbitrary sections so the **section** attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

constructor destructor

The **constructor** attribute causes the function to be called automatically before execution enters **main ()**. Similarly, the **destructor** attribute causes the function to be called automatically after **main ()** has completed or **exit ()** has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

These attributes are not currently implemented for Objective-C.

unused This attribute, attached to a function, means that the function is meant to be possibly unused. GNU CC will not produce a warning for this function.

weak The **weak** attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

alias (“target”)

The **alias** attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () {
    /* do something */;
}

void f () __attribute__ ((weak, alias (“__f”)));
```

declares **f** to be a weak alias for **__f**. In C++, the mangled name for the target must be used.

regparm (*number***)**

On the Intel 386, the **regparm** attribute causes the compiler to pass up to *number* integer arguments in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments will continue to be passed all of their arguments on the stack.

stdcall On the Intel 386, the **stdcall** attribute causes the compiler to assume that the called function will pop off the stack space used to pass arguments, unless it takes a variable number of arguments.

stdcall On the Intel 386, the **stdcall** attribute causes the compiler to assume that the called function will pop off the stack space used to pass arguments, unless it takes a variable number of arguments. This is useful to override the effects of the **stdcall** switch.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

Some people object to the **stdcall** feature, suggesting that ANSI C’s **#pragma** should be used instead. There are two reasons for not doing this.

1 It is impossible to generate **#pragma** commands from a macro.

2 There is no telling what the same **#pragma** might mean in another compiler.

These two reasons apply to almost any application that might be proposed for **#pragma**. It is basically a mistake to use **#pragma** for *anything*.

Prototypes and Old-Style Function Definitions

GNU C extends ANSI C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```
/* Use prototypes unless the compiler is old-fashioned. */
#if __STDC__
#define P(x) x
#else
#define P(x) ()
#endif
/* Prototype function declaration. */
int isroot P((uid_t));
/* Old-style function definition. */
int isroot (x)
/* ??? lossage here ??? */
uid_t x; {
    return x == 0;
}
```

Suppose the type **uid_t** happens to be **short**. ANSI C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an **int**, which does not match the prototype argument type of **short**.

This restriction of ANSI C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the **uid_t** type is **short**, **int**, or **long**. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion. Thus in GNU C the above example is equivalent to the following:

```
int isroot (uid_t);
int isroot (uid_t x) {
    return x == 0;
}
```

GNU C++ does not support old-style function definitions, so this extension is irrelevant.

C++ Style Comments

In GNU C, you may use C++ style comments, which start with **//** and continue until the end of the line. Many other C implementations allow such comments, and they are likely to be in a future C standard. However, C++ style comments are not recognized if you specify **-ansi** or **-traditional**, since they are incompatible with traditional constructs like **dividend//*comment*/divisor**.

Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

On some machines, dollar signs are allowed in identifiers if you specify **-traditional**. On a few systems they are allowed by default, even if you do not use **-traditional**. But they are never allowed if you specify **-ansi**.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$ lose (test)
```

The Character ESC in Constants

You can use the sequence **\e** in a string or character constant to stand for the ASCII character ESC.

Inquiring on Alignment of Types or Variables

The keyword **__alignof__** allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like **sizeof**.

For example, if the target machine requires a **double** value to be aligned on an 8-byte boundary, then **__alignof__ (double)** is 8. This is true on many RISC machines. On more traditional machine designs, **__alignof__ (double)** is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, **__alignof__** reports the *recommended* alignment of a type.

When the operand of **__alignof__** is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo {
    int x;
    char y;
} foo1;
```

the value of **__alignof__ (foo1.y)** is probably 2 or 4, the same as **__alignof__ (int)**, even though the data type of **foo1.y** does not itself demand any alignment.

A related feature which lets you specify the alignment of an object is **__attribute__ ((aligned (***alignment***))**; see the following section.

Specifying Attributes of Variables

The keyword **__attribute__** allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Eight attributes are currently defined for variables: **aligned**, **mode**, **nocommon**, **packed**, **section**, **transparent_union**, **unused**, and **weak**. Other attributes are available for functions (see “Declaring Attributes of Functions”) and for types (see “Specifying Attributes of Types”).

You may also specify attributes with **__** preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use **__aligned__** instead of **aligned**.

aligned (*alignment***)**

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable **x** on a 16-byte boundary. On a 68040, this could be used in conjunction with an **asm** expression to access the **move16** instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned **int** pair, you could write:

```
struct foo {
    int x[2] __attribute__ ((aligned (8)));
};
```

This is an alternative to creating a union with a **double** member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an **aligned** attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The **aligned** attribute can only increase the alignment; but you can decrease it by specifying **packed** as well. See below.

Note that the effectiveness of **aligned** attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying **aligned(16)** in an **__attribute__** will still only provide you with 8 byte alignment. See your linker documentation for further information.

mode (*mode***)**

This attribute specifies the data type for the declaration—whichever type corresponds to the mode MODE. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of **byte** or **__byte__** to indicate the mode corresponding to a one-byte integer, **word** or **__word__** for the mode of a one-word integer, and **pointer** or **__pointer__** for the mode used to represent pointers.

nocommon

This attribute specifies requests GNU CC not to place a variable "common" but instead to allocate space for it directly. If you specify the **-fno-common** flag, GNU CC will do this for all variables.

Specifying the **nocommon** attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

packed The **packed** attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the **aligned** attribute.

Here is a structure in which the field **x** is packed, so that it immediately follows **a**:

```
struct foo {
    char a;
    int x[2] __attribute__ ((packed));
};
```

section ("section-name")

Normally, the compiler places the objects it generates in sections like **data** and **bss**. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The **section** attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0
};
struct duart b __attribute__ ((section ("DUART_B"))) = { 0
};
char stack[10000] __attribute__ ((section ("STACK"))) = { 0
};
```

```
int init_data_copy __attribute__ ((section
(“INITDATACOPY”))) = 0;

main() {
    /* Initialize stack pointer */
    init_sp (stack + sizeof (stack));

    /* Initialize initialized data */
    memcpy (&init_data_copy, &data, &edata - &data);

    /* Turn on the serial ports */
    init_duart (&a);
    init_duart (&b);
}
```

Use the **section** attribute with an *initialized* definition of a *global* variable, as shown in the example. GNU CC issues a warning and otherwise ignores the **section** attribute in uninitialized variable declarations.

You may only use the **section** attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the **common** (or **bss**) section and can be multiply “defined”. You can force a variable to be initialized with the **-fno-common** flag or the **nocommon** attribute.

Some file formats do not support arbitrary sections so the **section** attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

transparent_union
This attribute, attached to a function argument variable which is a union, means to pass the argument in the same way that the first union member would be passed. You can also use this attribute on a **typedef** for a union data type; then it applies to all function arguments with that type.

unused This attribute, attached to a variable, means that the variable is meant to be possibly unused. GNU CC will not produce a warning for this variable.

weak The **weak** attribute is described in “Declaring Attributes of Functions”.

To specify multiple attributes, separate them by commas within the double parentheses: for example, **__attribute__ ((aligned (16), packed))**.

Specifying Attributes of Types

The keyword **__attribute__** allows you to specify special attributes of **struct** and **union** types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Three attributes are currently defined for types: **aligned**, **packed**, and **transparent_union**. Other attributes are defined for functions (see "Declaring Attributes of Functions") and for variables (see "Specifying Attributes of Variables").

You may also specify any one of these attributes with **__** preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use **__aligned__** instead of **aligned**.

You may specify the **aligned** and **transparent_union** attributes either in a **typedef** declaration or just past the closing curly brace of a complete enum, struct or union type *definition* and the **packed** attribute only past the closing brace of a definition.

aligned (*alignment***)**

This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
struct S {
    short f[3];
} __attribute__ ((aligned (8));

typedef int more_aligned_int __attribute__ ((aligned (8));
```

force the compiler to insure (as fas as it can) that each variable whose type is **struct S** or **more_aligned_int** will be allocated and aligned *at least* on a 8-byte boundary. On a Sparc, having all variables of type **struct S** aligned to 8-byte boundaries allows the compiler to use the **ldd** and **std** (doubleword load and store) instructions when copying one variable of type **struct S** to another, thus improving run-time efficiency.

Note that the alignment of any given **struct** or **union** type is required by the ANSI C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the **struct** or **union** in question. This means that you *can* effectively adjust the alignment of a **struct** or **union** type by attaching an **aligned** attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire **struct** or **union** type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given **struct** or **union** type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S {
    short f[3];
} __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an **aligned** attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each **short** is 2 bytes, then the size of the entire **struct S** type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire **struct S** type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The **aligned** attribute can only increase the alignment; but you can decrease it by specifying **packed** as well. See below.

Note that the effectiveness of **aligned** attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying **aligned(16)** in an **__attribute__** will still only provide you with 8 byte alignment. See your linker documentation for further information.

packed This attribute, attached to an **enum**, **struct**, or **union** type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for **struct** and **union** types is equivalent to specifying the **packed** attribute on each of the structure or union members. Specifying the **-fshort-enums** flag on the line is equivalent to specifying the **packed** attribute on all **enum** definitions.

You may only specify this attribute after a closing curly brace on an **enum** definition, not in a **typedef** declaration.

transparent_union

This attribute, attached to a **union** type definition, indicates that any variable having that union type should, if passed to a function, be passed in the same way that the first union member would be passed. For example:

```
union foo {
    char a;
    int x[2];
} __attribute__ ((transparent_union));
```

To specify multiple attributes, separate them by commas within the double parentheses: for example, **__attribute__ ((aligned (16), packed))**.

An Inline Function is As Fast As a Macro

By declaring a function **inline**, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use **-O**, no function is really inline.

To declare a function inline, use the **inline** keyword in its declaration, like this:

```
inline int inc (int *a) { (*a)++; }
```

(If you are writing a header file to be included in ANSI C programs, write **__inline__** instead of **inline**. See "Alternate Keywords")

You can also make all “simple enough” functions inline with the option **-finline-functions**. Note that certain usages in a function definition can make it unsuitable for inline substitution.

Note that in C and Objective-C, unlike C++, the **inline** keyword does not affect the linkage of the function.

GNU CC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared **inline**. (You can override this with **-fno-default-inline**; See “Options Controlling C++ Dialect”)

When a function is both inline and **static**, if all calls to the function are integrated into the caller, and the function’s address is never used, then the function’s own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option **-fkeep-inline-functions**. Some calls cannot be integrated for various reasons (in particular, calls that precede the function’s definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can’t be inlined.

When an inline function is not **static**, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-**static** inline function is always compiled on its own in the usual fashion.

If you specify both **inline** and **extern** in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of **inline** and **extern** has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking **inline** and **extern**) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but GNU found that a correct implementation when not optimizing was difficult. So they turned it off.

Assembler Instructions with C Expression Operands

In an assembler instruction using **asm**, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's **fsinx** instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here **angle** is the C expression for the input operand while **result** is that of the output operand. Each has **"f"** as its operand constraint, saying that a floating point register is required. The **=** in **=f** indicates that the operand is an output; all output operands constraints must use **=**. The constraints use the same language used in the machine description.

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended **asm** feature is most often used for machine instructions that the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit field), your constraint must allow a register. In that case, GNU CC will use the register as the output of the **asm**, and then store that register into the output.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character **+**, which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between

them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) **combine** instruction with **bar** as its read-only source operand and **foo** as its read-write destination:

```
asm (“combine %2,%0” : “=r” (foo) : “0” (foo), “g” (bar));
```

The constraint **”0”** for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that **foo** is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm (“combine %2,%0” : “=r” (foo) : “r” (foo), “g” (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of **foo** in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to **foo**’s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can’t tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile (“movc3 %0,%1,%2” : /* no outputs */ : “g” (from), “g”
(to), “g” (count) : “r0”, “r1”, “r2”, “r3”, “r4”, “r5”);
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register’s value is modified. In many assemblers, the register names begin with **%**; to produce one **%** in the assembler code, you must write **%%** in the input.

If your assembler instruction can alter the condition code register, add **cc** to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; **cc** serves to name this register. On other machines, the condition code is handled differently, and specifying **cc** has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add **memory** to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single **asm** template, separated either with newlines (written as **\n**) or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine **_foo** accepts arguments in registers 9 and 10:

```
asm (“movl %0,r9;movl %1,r10;call _foo” : /* no outputs */ : “g”
(from), “g” (to) : “r9”, “r10”);
```

Unless an output operand has the **&** constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use **&** for each output operand that may not overlap an input.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the **asm** construct, as follows:

```
asm (“clr %0;frob %1;beq 0f;mov #1,%0;0:” : “g” (result) : “g”
(input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one **asm** to another are not supported. The compiler’s optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these **asm** instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x) \
({ double __value, __arg = (x); \
    asm (“fsinx %1,%0”: “=f” (__value): “f” (__arg)); \
    __value;
})
```

Here the variable **__arg** is used to make sure that the instruction operates on a proper **double** value, and to accept only those arguments **x** which can convert automatically to a **double**.

Another way to make sure the instruction operates on the correct data type is to use a cast in the **asm**. This is different from using a variable **__arg** in that it converts more different types. For example, if the desired type were **int**, casting the argument to **int** would accept a pointer with no complaint, while assigning the argument to an **int** variable named **__arg** would warn about using a pointer unless the caller explicitly casts it.

If an **asm** has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an **asm** instruction from being deleted, moved significantly, or combined, by writing the keyword **volatile** after the **asm**. For example:

```
#define set_priority(x) \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile **asm** instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile **asm** instructions to remain perfectly consecutive. If you want consecutive output, use a single **asm**.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when GNU attempted to implement this, they found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write **__asm__** instead of **asm**. See "Alternate Keywords".

Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the **asm** (or **__asm__**) keyword after the declarator as follows:

```
int foo asm (“myfoo”) = 2;
```

This specifies that the name to be used for the variable **foo** in the assembler code should be **myfoo** rather than the usual **_foo**.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use **asm** in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting **asm** there, like this:

```
extern func () asm (“FUNC”);
func (x, y) int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler’s data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses.
- These local variables are sometimes convenient for use with the extended **asm** feature See “Assembler Instructions with C Expression Operands” if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the **asm**.)

Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm (“a5”);
```

Here **a5** is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register **a5** would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a “global” register that is not affected magically by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register **%a5**.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function **foo** by way of a third function **lose** that was compiled without knowledge of this variable (that is, in a different source file in which the variable wasn’t declared). This is because **lose** might save the register and put some other value there. For example, you can’t expect a global register variable to be available in the comparison-function that you pass to **qsort**, since **qsort** might have put something else in that register. (If you are prepared to recompile **qsort** with the same global register variable, you can solve this problem.)

If you want to recompile **qsort** or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option **-ffixed-REG**. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part

of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, **longjmp** will restore to each global register variable the value it had at the time of the **setjmp**. On some machines, however, **longjmp** will not change the value of global register variables. To be portable, the function that called **setjmp** should make other arrangements to save the values of the global register variables, and to restore them in a **longjmp**. This way, the same thing will happen regardless of what **longjmp** does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as **getwd**, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm (“a5”);
```

Here **a5** is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see “Assembler Instructions with C Expression Operands”). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register **%a5**.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable’s value is not live. However, these

registers are made unavailable for use in the reload pass. Excessive use of this feature may leave the compiler too few available registers to compile certain functions.

Alternate Keywords

The option **-traditional** disables certain keywords; **-ansi** disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords **asm**, **typeof** and **inline** cannot be used since they won't work in a program compiled with **-ansi**, while the keywords **const**, **volatile**, **signed**, **typeof** and **inline** won't work in a program compiled with **-traditional**.

The way to solve these problems is to put **__** at the beginning and end of each problematical keyword. For example, use **__asm__** instead of **asm**, **__const__** instead of **const**, and **__inline__** instead of **inline**.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

-pedantic causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing **__extension__** before the expression. **__extension__** has no effect aside from this.

Incomplete enum Types

You can define an **enum** tag without specifying its possible values. This results in an incomplete type, much like what you get if you write **struct foo** without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of **enum** more consistent with the way **struct** and **union** are handled.

This extension is not supported by GNU C++.

Function Names as Strings

GNU CC predefines two string variables to be the name of the current function. The variable **__FUNCTION__** is the name of the function as it appears in the source. The variable **__PRETTY_FUNCTION__** is the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

```
extern “C” {
    extern int printf (char *, ...);
}

class a {
    public: sub (int i) {
       printf (“__FUNCTION__ = %s\n”, __FUNCTION__);
       printf (“__PRETTY_FUNCTION__ = %s\n”, __PRETTY_FUNCTION__);
    }
};

int main (void) {
    a ax;
    ax.sub (0);
    return 0;
}
```

gives this output:

```
__FUNCTION__ = sub __PRETTY_FUNCTION__ = int a::sub (int)
```

These names are not macros: they are predefined string variables. For example, **#ifdef __FUNCTION__** does not have any special meaning inside a function, since the preprocessor does not do anything special with the identifier **__FUNCTION__**.

C++ Programming Notes

This section contains miscellaneous notes about programming in C++ with NeXT’s version of the GNU C++ compiler.

Multiple Virtual Inheritance

The C++ compiler invokes virtual functions correctly—except when a non-virtual function is redeclared as virtual in a subclass. The compiler issues a warning in this case, however.

In this example, the function **f()** in class **Animal** is redeclared virtual in the subclass **Mammal**:

```
class Animal{ void f(); }
class Mammal : public virtual Animal{ virtual void f(); }
class Quadruped : public virtual Animal{ virtual void f(); }
class Dog : public Mammal, public Quadruped{ virtual void f(); }
class Terrier : public Dog{ virtual void f(); }
```

Invoking the method **f()** gives the wrong result in the following case:

```
void zoo(void) {
    Terrier* terrier = new Terrier;
    Mammal* mammal = terrier;
    Quadruped* quadruped = terrier;
    Dog* dog = terrier;

    quadruped->f();// Wrong - invokes Dog::f()
    mammal->f();// Right - invokes Terrier::f()
    dog->f();// Right - invokes Terrier::f()
}
```

The compiler warns that wrong code may be generated:

```
    warning: method `Animal::f()' redeclared as `virtual Mammal::f()'
```

If you modify the above hierarchy by making the function **f()** in **Animal** virtual, the invocation works correctly. The workaround is therefore to make **f()** virtual throughout the hierarchy:

```
class Animal{ virtual void f(); }
class Mammal : public virtual Animal{ virtual void f(); }
class Quadruped : public virtual Animal{ virtual void f(); }
class Dog : public Mammal, public Quadruped{ virtual void f(); }
class Terrier : public Dog{ virtual void f(); }
```

Pointers to Member Functions

The C++ compiler flags as an error the use of member function pointers with objects that might not recognize the pointer or its contents. Here's an example of such errors:

```
class Mammal { public: void f(int); };
class Cat : public Mammal { public: void f(int); };

void g (Cat* aCat, Mammal* aMammal) {
    void (Mammal::*mammal_f_ptr)(int) = &Mammal::f;
    void (Cat::*cat_f_ptr)(int) = &Cat::f;

    (aCat->*mammal_f_ptr)(4); // OK
    (aMammal->*cat_f_ptr)(5); // Error (1)
    cat_f_ptr = &Cat::f;  // OK
    mammal_f_ptr = &Cat::f;  // Error (2)
}
```

The local variables **mammal_f_ptr** and **cat_f_ptr** are pointers to member functions, and the function **g** initializes them to point to the **class Cat** member function **f**. It then attempts to invoke this function through these pointers. Statement (1) is an error because you can't be sure that a **Mammal** object, **aMammal**, “responds to” a **Cat** member pointer, **cat_f_ptr**—especially since **cat_f_ptr** points to a **Cat** member function that **Mammal** would know nothing about. Even if **cat_f_ptr** were initialized to a **Mammal** member function, **cat_f_ptr** cannot safely be applied to a **Mammal** object. The assignment in (2) is an error because you cannot be sure that some member function of a derived class (in this case **Cat::f**) is available in any of its base classes (in this case **Mammal**).

Implicit Cast From void* to C++ Object Pointer

The ANSI C++ standard doesn’t allow implicit casts from **void*** to any C++ object pointer type. When such a cast is detected, the C++ compiler issues a warning. For example, if **aClass** is some arbitrary class, the following implicit cast produces a warning:

```
void *vp1;
aClass *obj1, *obj2;
vp1 = &obj1;
obj2 = vp1;// Warning: implicitly casts void pointer
```

You can still explictly cast with:

```
obj2 = (aClass *)vp1;
```

Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro **__GNUC__**. You can also use **__GNUG__** to test specifically for GNU C++ (see *The GNU C Preprocessor* for more information).

Named Return Values in C++

GNU C++ extends the function-definition syntax to allow you to specify a name for the result of a function outside the body of the definition, in C++ programs:

```
TYPE FUNCTIONNAME (ARGS) return RESULTNAME; { ... BODY ... }
```

You can use this feature to avoid an extra constructor call when a function result has a class type. For example, consider a function **m**, declared as **X v = m ();**, whose result is of class **X**:

```
X m () {
    X b;
    b.a = 23;
    return b;
}
```

Although **m** appears to have no arguments, in fact it has one implicit argument: the address of the return value. At invocation, the address of enough space to hold **v** is sent in as the implicit argument. Then **b** is constructed and its **a** field is set to the value 23. Finally, a copy constructor (a constructor of the form **X(X&)**) is applied to **b**, with the (implicit) return value location as the target, so that **v** is now bound to the return value.

But this is wasteful. The local **b** is declared just to hold something that will be copied right out. While a compiler that combined an “elision” algorithm with interprocedural data flow analysis could conceivably eliminate all of this, it is much more practical to allow you to assist the compiler in generating efficient code by manipulating the return value explicitly, thus avoiding the local variable and copy constructor altogether.

Using the extended GNU C++ function-definition syntax, you can avoid the temporary allocation and copying by naming **r** as your return value at the outset, and assigning to its **a** field directly:

```
X m () return r;
{
    r.a = 23;
}
```

The declaration of **r** is a standard, proper declaration, whose effects are executed *before* any of the body of **m**.

Functions of this type impose no additional restrictions; in particular, you can execute **return** statements, or return implicitly by reaching the end of the function body ("falling off the edge"). Cases like

```
X m () return r (23);
{
    return;
}
```

(or even **X m () return r (23); { }**) are unambiguous, since the return value **r** has been initialized in either case. The following code may be hard to read, but also works predictably:

```
X m () return r;
{
    X b;
    return b;
}
```

The return value slot denoted by **r** is initialized at the outset, but the statement **return b;** overrides this value. The compiler deals with this by destroying **r** (calling the destructor if there is one, or doing nothing if there is not), and then reinitializing **r** with **b**.

This extension is provided primarily to help people who use overloaded operators, where there is a great need to control not just the arguments, but the return values of functions. For classes where the copy constructor incurs a heavy performance penalty (especially in the common case where there is a quick default constructor), this is a major savings. The disadvantage of this extension is that you do not control when the default constructor for the return value is called: it is always called at the beginning.

Minimum and Maximum Operators in C++

It is very convenient to have operators which return the "minimum" or the "maximum" of two arguments. In GNU C++ (but not in GNU C),

A <? B is the "minimum", returning the smaller of the numeric values A and B;

A >? B is the “maximum”, returning the larger of the numeric values A and B.

These operations are not primitive in ordinary C++, since you can use a macro to return the minimum of two things in C++, as in the following example.

```
#define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

You might then use **int min = MIN (i, j);** to set MIN to the minimum value of variables I and J.

However, side effects in **X** or **Y** may cause unintended behavior. For example, **MIN (i++, j++)** will fail, incrementing the smaller counter twice. A GNU C extension allows you to write safe macros that avoid this kind of problem (see “Naming an Expression’s Type”). However, writing **MIN** and **MAX** as macros also forces you to use function-call notation notation for a fundamental arithmetic operation. Using GNU C++ extensions, you can write **int min = i <? j;** instead.

Since **<?** and **>?** are built into the compiler, they properly handle expressions with side-effects; **int min = i++ <? j++;** works correctly.

“goto” and Destructors in GNU C++

In C++ programs, you can safely use the **goto** statement. When you use it to exit a block which contains aggregates requiring destructors, the destructors will run before the **goto** transfers control. (In ANSI C++, **goto** is restricted to targets within the current block.)

The compiler still forbids using **goto** to *enter* a scope that requires constructors.

Declarations and Definitions in One Header

C++ object definitions can be quite complex. In principle, your source code will need two kinds of things for each object that you use across more than one source file. First, you need an “interface” specification, describing its structure with type declarations and function prototypes. Second, you need the “implementation” itself. It can be tedious to maintain a separate interface description in a header file, in parallel to the actual implementation. It is also dangerous, since separate interface and implementation definitions may not remain parallel.

With GNU C++, you can use a single header file for both purposes.

Warning: The mechanism to specify this is in transition. For now, you must use one of two **#pragma** commands; in a future release of GNU C++, an alternative mechanism will make these **#pragma** commands unnecessary.

The header file contains the full definitions, but is marked with **#pragma interface** in the source code. This allows the compiler to use the header file only as an interface specification when ordinary source files incorporate it with **#include**. In the single source file where the full implementation belongs, you can use either a naming convention or **#pragma implementation** to indicate this alternate use of the header file.

#pragma interface
#pragma interface "*subdir/objects***.h"**

Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing **#pragma interface** is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses **#pragma implementation**'). Instead, the object files will contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to **#pragma implementation**.

#pragma implementation
#pragma implementation "*objects***.h"**

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use **#pragma interface**. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use **#pragma implementation** with no argument, it applies to an include file with the same basename as your source file. (A file's "basename" was the name stripped of all leading path information and of trailing suffixes, such as **.h** or **.C** or **.cc**.) For example, in **allclass.cc**, **#pragma implementation** by itself is equivalent to **#pragma implementation "allclass.h"**.

In versions of GNU C++ prior to 2.6.0 **allclass.h** was treated as an implementation file whenever you would include it from **allclass.cc** even if you never specified **#pragma implementation**. This was deemed to be more trouble than it was worth, however, and disabled.

If you use an explicit **#pragma implementation**, it must appear in your source file *before* you include the affected header files.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use **#include** to include the header file; **#pragma implementation** only specifies how to use the file—it doesn't actually include it.)

There is no way to split up the contents of a single header file into multiple implementation files.

#pragma implementation and **#pragma interface** also have an effect on function inlining.

If you define a class in a header file marked with **#pragma interface**, the effect on a function defined in that class is similar to an explicit **extern** declaration—the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as **#pragma implementation**, the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with **-fno-implement-inlines**. If any calls were not inlined, you will get linker errors.

#pragma cplusplus.

This pragma can be used to resolve the problem of having C++ system header files. All system header files are by default included in implicit **extern "C"**. When **#pragma cplusplus** appears in a header file, the rest of that file is embedded in an implicit **extern "C++"** block.

An error is reported if this pragma appears inside an **explicit extern "C" {...}**.

Type Abstraction using Signatures

In GNU C++, you can use the keyword **signature** to define a completely abstract class interface as a datatype. You can connect this abstraction with actual classes using signature pointers. If you want to use signatures, run the GNU compiler with the **-fhandle-signatures** command-line option. (With this option, the compiler reserves a second keyword **sigof** as well, for a future extension.)

Roughly, signatures are type abstractions or interfaces of classes, and are similar to Objective-C's protocols. Some other languages have similar facilities. C++ signatures are related to ML's signatures, Haskell's type classes, definition modules in Modula-2, interface modules in Modula-3, abstract types in Emerald, type modules in Trellis/Owl, categories in Scratchpad II, and types in POOL-I. For a more detailed discussion of signatures, see *Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++* by Gerald Baumgartner and Vincent F. Russo (Tech report CSD-TR-95-051, Dept. of Computer Sciences, Purdue University, August 1995, a slightly improved version appeared in *Software—Practice & Experience*, 25(8), pp. 863-889, August 1995). You can get the tech report by anonymous FTP from **ftp.cs.purdue.edu** in **pub/gb/Signature-design.ps.gz**.

Syntactically, a signature declaration is a collection of member function declarations and nested type declarations. For example, this signature declaration defines a new abstract type **S** with member functions **int foo ()** and **int bar (int)**:

```
signature S {
    int foo ();
    int bar (int);
};
```

Since signature types do not include implementation definitions, you cannot write an instance of a signature directly. Instead, you can define a pointer to any class that contains the required interfaces as a "signature pointer". Such a class "implements" the signature type.

To use a class as an implementation of **S**, you must ensure that the class has public member functions **int foo ()** and **int bar (int)**. The class can have other member functions as well, public or not; as long as it offers what's declared in the signature, it is suitable as an implementation of that signature type.

For example, suppose that **C** is a class that meets the requirements of signature **S** (**C** "conforms to" **S**). Then

```
C obj;
S * p = &obj;
```

defines a signature pointer **p** and initializes it to point to an object of type **C**. The member function call **int i = p->foo ();** executes **obj.foo ()**.

Abstract virtual classes provide somewhat similar facilities in standard C++. There are two main advantages to using signatures instead:

1. Subtyping becomes independent from inheritance. A class or signature type **T** is a subtype of a signature type **S** independent of any inheritance hierarchy as long as all the

member functions declared in **S** are also found in **T**. So you can define a subtype hierarchy that is completely independent from any inheritance (implementation) hierarchy, instead of being forced to use types that mirror the class inheritance hierarchy.

2. Signatures allow you to work with existing class hierarchies as implementations of a signature type. If those class hierarchies are only available in compiled form, you're out of luck with abstract virtual classes, since an abstract virtual class cannot be retrofitted on top of existing class hierarchies. So you would be required to write interface classes as subtypes of the abstract virtual class.

There is one more detail about signatures. A signature declaration can contain member function *definitions* as well as member function declarations. A signature member function with a full definition is called a *default implementation*; classes need not contain that particular interface in order to conform. For example, a class **C** can conform to the signature

```
signature T {
    int f (int);
    int f0 () {
       return f (0);
    };
};
```

whether or not **C** implements the member function **int f0 ()**. If you define **C::f0**, that definition takes precedence; otherwise, the default implementation **S::f0** applies.

Known Causes of Trouble with GNU CC

This section describes known problems that affect users of GNU CC. Most of these are not GNU CC bugs per se—if they were, GNU would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

Problems in the Compiler

- There are several obscure cases of mis-using struct, union, and enum tags that are not detected as errors by the compiler.

- When **-pedantic-errors** is specified, GNU C will incorrectly give an error message when a function name is specified in an expression involving the comma operator.

- Loop unrolling doesn't work properly for certain C++ programs. This is a bug in the C++ front end. It sometimes emits incorrect debug info, and the loop unrolling code is unable to recover from this error.

Interoperation

This section lists various difficulties encountered in using GNU C or GNU C++ together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

GNU C++ does not do name mangling in the same way as other C++ compilers. This means that object files compiled with one compiler cannot be used with another.

This effect is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation, including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers—but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.

- Older GDB versions sometimes fail to read the output of GNU CC version 2. If you have trouble, get GDB version 4.4 or later.

- Use of **-I/usr/include** may cause trouble.

- On a Sparc, GNU CC aligns all values of type **double** on an 8-byte boundary, and it expects every **double** to be so aligned. The Sun compiler usually gives **double** values 8-byte alignment, with one exception: function arguments of type **double** may not be aligned.

  As a result, if a function compiled with Sun CC takes the address of an argument of type **double** and passes this pointer of type **double *** to a function compiled with GNU CC, dereferencing the pointer may cause a fatal signal.

  One way to solve this problem is to compile your entire program with GNU CC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function **access_double** instead of directly with ***:

```
inline double access_double (double *unaligned_ptr) {
    union d2i {
       double d;
       int i[2];
    };
    union d2i *p = (union d2i *) unaligned_ptr;
    union d2i u;
    u.i[0] = p->i[0];
    u.i[1] = p->i[1];
    return u.d;
}
```

Storing into the pointer can be done likewise with the same union.

- On Solaris, the **malloc** function in the **libmalloc.a** library may allocate memory that is only 4 byte aligned. Since GNU CC on the Sparc assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the **libmalloc.a** library.

  The solution is to not use the **libmalloc.a** library. Use instead **malloc** and related functions from **libc.a**; they do not have this problem.

- The 128-bit long double format that the Sparc port supports currently works by using the architecturally defined quad-word floating point instructions. Since there is no hardware that supports these instructions they must be emulated by the operating system. Long doubles do not work in Sun OS versions 4.0.3 and earlier, because the kernel emulator uses an obsolete and incompatible format. Long doubles do not work in Sun OS version 4.1.1 due to a problem in a Sun library. Long doubles do work on Sun OS versions 4.1.2 and higher, but GNU CC does not enable them by default. Long doubles appear to work in Sun OS 5.x (Solaris 2.x).

- On HP-UX version 9.01 on the HP PA, the HP compiler **cc** does not compile GNU CC correctly. We do not yet know why. However, GNU CC compiled on earlier HP-UX versions works properly on HP-UX 9.01 and can compile itself properly on 9.01.

- On the HP PA machine, ADB sometimes fails to work on functions compiled with GNU CC. Specifically, it fails to work on functions that use **alloca** or variable-size arrays. This is because GNU CC doesn’t generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.

- Taking the address of a label may generate errors from the HP-UX PA assembler. GAS for the PA does not have this problem.

- Using floating point parameters for indirect calls to static functions will not work when using the HP assembler. There simply is no way for GCC to specify what registers hold

arguments for static functions when using the HP assembler. GAS for the PA does not have this problem.

- In extremely rare cases involving some very large functions you may receive errors from the HP linker complaining about an out of bounds unconditional branch offset. This used to occur more often in previous versions of GNU CC, but is now exceptionally rare. If you should run into it, you can work around by making your function smaller.
- GNU CC compiled code sometimes emits warnings from the HP-UX assembler of the form:

  ```
  (warning) Use of GR3 when frame >= 8192 may cause conflict.
  ```

  These warnings are harmless and can be safely ignored.

Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The **-traditional** option eliminates many of these incompatibilities, *but not all*, by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call **mktemp** with a string constant argument. The function **mktemp** always alters the string its argument points to.

  Another consequence is that **sscanf** does not work on some systems when passed a string constant as its format control string or input. This is because **sscanf** incorrectly tries to write into the string constant. Likewise **fscanf** and **scanf**.

  The best solution to these problems is to change the program to use **char**-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the **-fwritable-strings** flag, which directs GNU CC to handle string constants the same way most C compilers do. **-traditional** also has this effect, among others.

- **-2147483648** is positive.

  This is because 2147483648 cannot fit in the type **int**, so (following the ANSI C rules) its data type is **unsigned long int**. Negating this value yields 2147483648 again.

- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) “a”
  ```

  will produce output **”a”** regardless of what the argument A is.

- The **-traditional** option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use **setjmp** and **longjmp**, the only automatic variables guaranteed to remain valid are those declared **volatile**. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo () {
      int a, b;
      a = fun1 ();
      if (setjmp (j))
         return a;
      a = fun2 ();

      /* longjmp (j) may occur in fun3. */
      return a + fun3 ();
  }
  ```

  Here **a** may or may not be restored to its first value when the **longjmp** occurs. If **a** is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

  If you use the **-W** option with the **-O** option, you will get a warning when GNU CC thinks such a problem might be possible.

  The **-traditional** option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call **setjmp**. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GNU CC. For example, a program like this will not work:

  ```
  foobar ( #define luser hack)
  ```

  ANSI C does not permit such a construct, and neither does GNU CC—even with **-traditional**.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a **extern** declaration affects all the rest of the file even if it happens within a block.

  The **-traditional** option directs GNU C to treat all **extern** declarations as global, like traditional compilers.

- In traditional C, you can combine **long**, etc., with a typedef name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ANSI C, this is not allowed: **long** and other type modifiers require an explicit **int**. Because this criterion is expressed by Bison grammar rules rather than C code, the **-traditional** flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.

- PCC allows whitespace in the middle of compound assignment operators such as `+=`. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.

- GNU CC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
      You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by **/*...*/**. However, **-traditional** suppresses these error messages.

- Many user programs contain the declaration **long time ();**. In the past, the system header files on many systems did not actually declare **time**, so it did not matter what type your program declared it to return. But in systems with ANSI C headers, **time** is declared to return **time_t**, and if that is not the same as **long**, then **long time ();** is erroneous.

  The solution is to change your program to use **time_t** as the return type of **time**.

- When compiling functions that return **float**, PCC converts it to a double. GNU CC actually returns a **float**. If you are concerned with PCC compatibility, you should declare your functions to return **double**; you might as well say what you mean.

- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros **STRUCT_VALUE** and **STRUCT_INCOMING_VALUE** tell GNU CC where to pass this address.

  By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GNU CC does not use this method because it is slower and nonreentrant.

- On some newer machines, PCC uses a reentrant convention for all structure and union returning. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

  You can tell GNU CC to use a compatible convention for all structure and union returning with the option **-fpcc-struct-return**.

- GNU C complains about program fragments such as **0x74ae-0x4000** which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single “preprocessing token”. Each such token must correspond to one token in C. Since this does not, GNU C prints an error message. Although it may appear obvious that what is meant is an operator and two values, the ANSI C standard specifically requires that this be treated as erroneous.

  A “preprocessing token” is a “preprocessing number” if it begins with a digit and is followed by letters, underscores, digits, periods and **e+**, **e-**, **E+**, or **E-** character sequences.

  To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GNU CC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GNU CC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- Users often think it is a bug when GNU CC reports an error for code like this:

  ```
  int foo (struct mumble *);
  struct mumble {
      ...
  };
  int foo (struct mumble *x) {
      ...
  }
  ```

  This code really is erroneous, because the scope of **struct mumble** in the prototype is limited to the argument list containing it. It does not refer to the **struct mumble** defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But in the definition of **foo**, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

  This behavior may seem silly, but it's what the ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of **struct mumble** above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

- Accesses to bitfields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bitfield; it may even vary for a given bitfield according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bitfields.

- On 68000 and **i386** systems, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the floating point registers hold a few more bits of precision than fit in a **double** in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the **-ffloat-store** or **-ffppc** options (see "Options That Control Optimization" earlier in this document).

Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ANSI C++ draft standard) is also evolving. As a result, your C++ compiler may occasionally surprise you, even when its behavior is correct. This section discusses some areas that frequently give rise to questions of this sort.

Declare *and* Define Static Members

When a class has static data members, it is not enough to *declare* the static member; you must also *define* it. For example:

```
class Foo {
    ...
    void method();
    static int bar;
};
```

This declaration only establishes that the class **Foo** has an **int** named **Foo::bar**, and a member function named **Foo::method**. But you still need to define *both* **method** and **bar** elsewhere. According to the draft ANSI standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to **g++** from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: **g++** reports as undefined symbols any static data members that lack definitions.

Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like the libg++ String class, that define a conversion function to type char * or const char *. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function **strfunc** that returns **String** objects, and another function **charfunc** that operates on pointers to **char**:

```
String strfunc ();
void charfunc (const char *);
```

In this situation, it may seem natural to write **charfunc (strfunc ());** based on the knowledge that class **String** has an explicit conversion to **char** pointers. However, what really happens is akin to **charfunc (strfunc ().convert ());**, where the **convert** method is a function to do the same data conversion normally performed by a cast. Since the last use of the temporary **String** object is the call to the conversion function, the compiler may delete that object before actually calling **charfunc**. The compiler has no way of knowing that deleting the **String** object will invalidate the pointer. The pointer then points to garbage, so that by the time **charfunc** is called, it gets an invalid argument.

Code like this may run successfully under some other compilers, especially those that delete temporaries relatively late. However, the GNU C++ behavior is also standard-conforming, so if your program depends on late destruction of temporaries it is not portable.

If you think this is surprising, you should be aware that the ANSI C++ committee continues to debate the lifetime-of-temporaries problem.

For now, at least, the safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
String& tmp = strfunc ();
charfunc (tmp);
```

Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

Errors report problems that make it impossible to compile your program. GNU CC reports errors with the source file name and line number where the problem is apparent.

Warnings report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text **warning:** to distinguish them from error messages.

Warnings may indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU C or C++. Many warnings are issued only if you ask for them, with one of the **-W** options (for instance, **-Wall** requests a variety of useful warnings).

GNU CC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The **-pedantic** option tells GNU CC to issue warnings in such cases; **-pedantic-errors** says to make them errors instead. This does not mean that *all* non-ANSI constructs get warnings or errors.

See “Options to Request or Suppress Warnings” for more detail on these and related command-line options.

Legal Considerations

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

   You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent

obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be

guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

13. END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

ONE LINE TO GIVE THE PROGRAM'S NAME AND A BRIEF IDEA OF WHAT IT DOES.

Copyright (C) 19*YY NAME OF AUTHOR*

This program is free software; you can redistribute it and/or modify it under the terms of 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. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19YY NAME OF AUTHOR Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type **show w**. This is free software, and you are welcome to redistribute it under certain conditions; type **show c** for details.

The hypothetical commands **show w** and **show c** should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than **show w** and **show c**; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program **Gnomovision** (which makes passes at compilers) written by James Hacker.

SIGNATURE OF TY COON, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.