11

The GNU C Compiler

The C compiler used on NEXTSTEP computers is GNU CC, an ANSI-standard C compiler produced by the Free Software Foundation. This compiler has been modified and extended as a compiler for the Objective C language by NeXT Computer, Inc. for use on NEXTSTEP computers. This chapter describes how to compile a C program using the GNU compiler.

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

The following sections describe command options available when compiling a C program with GNU CC, incompatibilities between GNU CC and non-ANSI versions of C, GNU extensions to the C language, and implementation-specific details related to using C on a NeXT computer.

GNU CC Command Options

When you invoke GNU CC with the **cc** command, it normally performs the following operations in the order shown here:

- Preprocessing (**cpp**)
- Compilation (**cc1**)
- Assembly (**as**)
- Linking (**ld**)

Some options described below 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 aren't documented here, since you rarely need to use any of them.

The GNU C compiler uses a command syntax much like the UNIX C compiler. The **gcc** program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: **-dr** is very different from **-d -r**.

Many options have long names starting with **-f** (**-fforce-mem**, **-fstrength-reduce**, 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 formsÐwhichever one *isn't* the default.

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; the fourth stageÐlinkingÐcombines 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. For example, a ª.cº file is C source code which must be preprocessed, a ª.iº file is C source code which shouldn't be preprocessed, a ª.ccº file is C++ source code which must be preprocessed, a ª.sº file is assembler code, and an unrecognized file name is considered an object file and is fed straight into linking.

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

-x *language*
Specify that the following input files are in the language *language*. This option applies to all following input files until the next **-x** option. Possible values of language are **c**, **objective-c**, **c-header**, **c++**, **cpp-output**, **assembler**, and **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).

The point at which the compilation process stops is controlled by various options:

-c Compile or assemble the source files, but don't link. The linking stage simply isn't 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 ª.cº, ª.iº, ª.sº, and so on, with ª.oº. Unrecognized input files, not requiring compilation or assembly, are ignored.

-S Stop after the stage of compilation proper; don't 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 ª.cº, ª.iº, etc., with ª.sº. Input files that don't require compilation are ignored.

-E Stop after the preprocessing stage; don't 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 doesn't make sense to use **-o** when compiling more than one input file, unless you are producing an executable file as output.)

If **-o** isn't 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 the standard output.

-v Print (to 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.

-vspec Print (to standard error output) all spec's processed by the **do_spec_1()** function in **gcc.c**. Also, print the commands executed to run the stages of compilation and version numbers, like the **-v** option.

-pipe Use pipes rather than temporary files for communication between the various stages of compilation.

-B*path* Compiler driver program tries *path* (which must end in /) as the directory prefix for each program it tries to run. These programs are **cpp**, **cc1**, **as**, and **ld**.

For each subprogram to be run, the compiler driver first tries the **-B** prefix, if any. If that name isn't found, or if **-B** wasn't specified, the driver tries two standard prefixes, **/bin/** and **/lib/**. If neither of those results in a file name that's found, the unmodified program name is searched for using the directories specified in your PATH environment variable.

Specifying a Dialect of the C Language

The following options control the dialect of C that the compiler accepts:

-ansi Support all ANSI 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.

The alternate keywords **__asm__**, **__extension__**, **__inline__**, and **__typeof__** continue to work despite **-ansi**. You wouldn't want to use them in an ANSI C program, of course, but it 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 doesn't cause non-ANSI C programs to be rejected gratuitously. For that, **-pedantic** is required in addition to **-ansi**. See the section Requesting or Suppressing 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.

-ObjC Compile a source file that contains Objective C language code (the file can have either a ª.cº or an ª.mº extension).

-bsd Enforce strict BSD semantics. When the **-bsd** option is used, the macro **__STRICT_BSD__** is predefined in the preprocessor. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros.

-trigraphs Support ANSI C trigraphs. 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 a function definition. This includes implicit declarations of functions.
- The keywords **typeof**, **inline**, **signed**, **const**, and **volatile** aren't 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 aren't an error.
- String ªconstantsº aren't 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.

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

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

- The predefined macro **__STDC__** isn't defined when you use **-traditional**, but **__GNUC__** is (since the GNU extensions which **__GNUC__** indicates aren't 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.

-fno-asm Don't recognize **asm**, **inline** or **typeof** as a keyword. These words may then be used as identifiers. You can use **__asm__**, **__inline__**, and **__typeof__** instead. **-ansi** implies **-fno-asm**.

-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 the unsigned, like **unsigned char**. (Note that each type of computer has a default for what **char** should beÐit's either like **unsigned char** by default or like **signed char** by default. The type **char** is always a distinct type from either **signed char** or **unsigned char**, even though its behavior is always just like one of those two.)

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.

-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, **-fno-signed-char** is equivalent to **-funsigned-char**.

-fsigned-bitfields, -funsigned-bitfields, -fno-signed-bitfields, -fno-unsigned-bitfields
Similar to the above flags, these options control whether a bitfield is signed or unsigned, when the declaration doesn't 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 **-traditiona**l 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. **-traditional** also has this effect. (Note that writing into string constants is a bad idea; [a]constants[o] should be constant.)

Requesting or Suppressing Warnings

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

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

-w Inhibit all warning messages.

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

Valid ANSI 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. There's no reason to use this option; it exists only to satisfy pedants.

-pedantic doesn't 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.

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

-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's 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;
}
```

Spurious warnings can occur because GNU CC doesn't realize that certain functions (including **abort()** and **longjmp()**) will never return.

- An expression-statement contains no side effects.
- An unsigned value is compared against zero with > or <=.

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

-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 isn't **void**.

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

-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.

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

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

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

-Wuninitialized
Warn if 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 don't 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 don't 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 isn't smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here's 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's 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's set.

Some spurious warnings can be avoided if you declare as volatile all the functions you use that never return.

-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 aren't implied by **-Wall** because they warn about constructions that we

consider reasonable to use, on occasion, in clean programs.

-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 compilers.

-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.

-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-byte 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.

Preparing Your Program for Debugging

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

-g Produce debugging information for use with GDB.

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.

-pg Generate extra code to write profile information suitable for the analysis program **gprof**.

-d*letters* 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:

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 *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**.

m Print statistics on memory usage to standard error, at the end of the run.

p Annotate the assembler output with a comment indicating which pattern and alternative was used.

-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 ªtemporaryº 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.cpp** and **foo.s**, as well as **foo.o**.

Controlling Optimization

These options control various sorts of optimizations:

-O 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. Also, only variables declared **register** are allocated in registers.

With **-O**, the compiler tries to reduce code size and execution time; also, **-fthread-jumps** and **-fdelayed-branch** are turned on.

-O2 Highly optimize. All supported optimizations that don't involve a space-speed tradeoff are performed. As compared to **-O**, this option will increase both compilation time and the performance of the generated code. All **-fflag** options that control optimization are turned on when **-O2** is specified.

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 listedÐthe one which *isn't* the default. You can figure out the other form by either removing **no-** or adding it.

-ffloat-store
Don't store floating point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a double is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use **-ffloat-store** for such programs.

-fno-defer-pop
Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) 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 may produce better code by making all memory references potential common subexpressions. When they aren't common subexpressions, instruction combination should eliminate the separate register-load.

-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 most machines.

-finline Pay attention to the **inline** keyword. Normally the negation of this option **-fno-inline** is used to keep the compiler from expanding any functions inline. However, the opposite effect may be desirable when compiling with **-g**, since **-g** normally turns off all inline function expansion.

-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.

-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.

-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
Don't 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 isn't used.

The following options control specific optimizations. The **-O2** option turns on all of these optimization except **-funroll-loops** and **-funroll-all-loops**. The **-O** option usually turns on the **-fthread-jumps** and **-fdelayed-branch** options, but specific machines may change the default optimizations.

You can use the following flags in the rare cases when ªfine-tuningº 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 we check 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.

-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-all-loops
Perform the optimization of loop unrolling. This is done for all loops. This usually makes programs run more slowly.

-fcse-follow-jumps
In common subexpression elimination, scan through jump instructions in certain cases. This isn't as powerful as completely global CSE, but not as slow either.

-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

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.

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.

-i *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 **-i** *file* is to make the macros defined in *file* available for use in the main input.

-nostdinc Don't 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 the section Specifying Directories to be Searched for more information on **-I**.

Between **-nostdinc** and **-I-**, you can eliminate all directories except those specified explicitly from the search path for header files.

-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** commands. 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 files **#include**d in it. This rule may be a single line or may be continued with backslash-newline if it's long. The list of rules is printed on standard output instead of the preprocessed C program.

-M implies **-E**.

-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 files with names made by replacing ª.cº with ª.dº at the end of the input file names. This is in addition to compiling the file as specifiedÐnote that **-MD** doesn't inhibit ordinary compilation the way **-M** does.

The Mach utility **md** can be used to merge the ª.dº 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.

-H Print the name of each header file used, in addition to other normal activities.

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

-D*macro*=*defn*
Define macro *macro* as *defn*.

-U*macro* Undefine macro *macro*.

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

Linking

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

object-file-name
A file name that doesn't 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 isn't run, and object file names shouldn't be used as arguments. See the section Controlling the Kind of Output for more information.

-nostdlib Don't use the standard system libraries and startup files when linking. Only the files you specify will be passed to the linker.

Additional linker options are described in the **ld**(1) UNIX manual page.

Specifying Directories to be Searched

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

-I*dir* Search directory *dir* for include files.

-I- Any directories specified with **-I** options before the **-I-** option are searched only for the case of **#include "***file***"**; they aren't 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's 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 isn't exactly the same as what the preprocessor does by default, but it's often satisfactory.

-I- doesn't 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**.

Specifying 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 listedÐthe one which *isn't* the default. You can figure out the other form by either removing **no-** or adding it.

-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.

-fno-common
Alocate 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'll get an error when you link them.

-fvolatile Consider all memory references through pointers to be volatile.

-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 doesn't 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 don't live across a call. Functions compiled this way won't 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 doesn't 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 doesn't have a negative form, because it specifies a three-way choice.

C Programming Notes

This section contains miscellaneous notes about programming in C on a NEXTSTEP computer. 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 belowÐthey 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.

[a]Suggested Reading[o] 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.

GNU Extensions to the C Language

GNU C provides several language features not found in ANSI 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.

Note: You should avoid the use of these GNU C extensions to the ANSI C language, since they aren't guaranteed to be supported in future releases of NEXTSTEP.

Casts as Lvalues

In GNU C, casts are allowed as lvalues provided their operands are lvalues. This means that you can store values into them.

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 wouldn't 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 inconsistancy, we think it's 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**.

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**.

Arithmetic on void-Pointers 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

The elements of an aggregate initializer for an automatic variable aren't required to be constant expressions in GNU C. Here's 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's 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 aren't simple constants aren't very useful, because the constructor isn't 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's 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.

Declaring Attributes of Functions

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

A few functions, such as **abort()** and **exit()**, cannot return. These functions should be declared **volatile**. For example,

```
extern void volatile abort ();
```

tells the compiler that it can assume that **abort()** won't return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables. It doesn't make sense for a volatile function to return anything other than **void**.

Many functions don't 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 **const**. For example,

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

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

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 doesn't make sense for a **const** function to return **void**.

We recommend placing the keyword **const** after the function's return type. It makes no difference in the example above, but when the return type is a pointer, it's the only way to make the function itself **const**. For example,

```
const char *mincp (int);
```

says that **mincp()** returns **const char *Đ**a pointer to a **const** object. To declare **mincp()** as **const**, you must write this:

```
char * const mincp (int);
```

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.

Dollar signs are allowed on certain machines if you specify **-traditional**. On a few systems they are allowed by default, even if **-traditional** isn't used. 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

In GNU C, you can use the sequence **\E** in a string or character constant to stand for the ASCII character ESC.

Specifying Attributes of Variables

In GNU C, the keyword **__attribute__** allows you to specify special attributes of variables or structure fields. The only attributes currently defined are the **aligned** and **format** attributes.

The **aligned** attribute specifies the alignment of the variable or structure field. 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 68000, 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 isn't possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed.

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 first parameter of the **format** attribute determines how the format string is interpreted, and should be either **printf** or **scanf**. The second parameter specifies the number of the format string argument (starting from 1). The third parameter specifies the number of the first argument which should be checked against the format string. For functions where the arguments aren't 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 to **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 C library functions **printf()**, **fprintf()**, **sprintf()**, **scanf()**, **fscanf()**, **sscanf()**, **vprintf()**, **vfprintf()**, and **vsprintf()** whenever such warnings are requested (using **-Wformat**), so there's no need to modify the header file **stdio.h**.

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.

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**.

You can also make all ªsimple enoughº functions inline with the option **-finline-functions**. Note that certain usages in a function definition can make it unsuitable for inline substitution.

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 doesn't 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's 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 isn't static, 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, 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 hadn't 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.

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's 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 the maximum number of operands in any instruction pattern in the machine description.

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 doesn't parse the assembler instruction template and doesn't know what it means, or whether it's valid assembler input. The extended **asm** feature is most often used for machine instructions that the compiler itself doesn't know exist.

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** doesn't 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 the 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 isn't enough to guarantee that they will be in the same place in the generated assembler code. The following wouldn't 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 isn't even mentioned in the assembler code, the result won't work, but GNU CC can't tell that.

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.

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's 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");
```

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 that your assembler supports local labels, as the GNU assembler and most UNIX assemblers do.

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 doesn't 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 won't be deleted or moved significantly, regardless, unless it's 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's a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following ªstoreº 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 ªtestº and ªcompareº instructions because they don't have any output operands.

Additional Information about GNU CC

This section describes a few areas that commonly cause problems for users of GNU CC, and points out incompatibilities between GNU C and some other existing versions of C.

Known Causes of Trouble with GNU CC

Here are some of the things that have caused trouble for people installing or using GNU CC.

- Users often 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: this code really is erroneous, because the old-style non-prototype 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);
  ```

- Users often 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 really is erroneous, because 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 belowÐthey 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.

This behavior may seem silly, but it's what the ANSI standard specifies. It's 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.

Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The **-traditional** option eliminates most of these incompatibilitiesÐbut not allÐby 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()** doesn't 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. This is also true of **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 isn't 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.

· GNU CC doesn't substitute macro arguments when they appear inside 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 ();
  /* @r{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, 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'll 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.

- 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, an **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 isn't 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, doesn't allow this. The difficulty described immediately above applies here too.

- GNU CC will flag unterminated character constants inside preprocessor 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.

- 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**.

- 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's 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 doesn't use this method because it's 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.

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 softwareÐto 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 ªProgramº, below, refers to any such program or work, and a ªwork based on the Programº 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 ªmodificationº.) Each licensee is addressed as ªyouº.

   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:

4. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

5. 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.

6. 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.

7. 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:

8. 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,

9. 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,

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

16. 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.

17. 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.

18. Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and ªany 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.

19. 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

20. 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 ªAS ISº 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.

21. 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.

END OF TERMS AND CONDITIONS

Appendix: 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 ªcopyrightº 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., 675 Mass Ave, Cambridge, MA 02139, 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) 19*yy 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 itemsÐwhatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ªcopyright disclaimerº for the program, if necessary. Here's 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.