NAME

gcc, g++ – GNU project C and C++ Compiler (v2.6)

SYNOPSIS

gcc [*option*|*filename*]...
g++ [*option*|*filename*]...

WARNING

The information in this man page is an extract from the full documentation of the GNU C compiler, and is limited to the meaning of the options.

This man page is not kept up to date except when volunteers want to maintain it. If you find a discrepancy between the man page and the software, please check the Info file, which is the authoritative documentation.

If we find that the things in this man page that are out of date cause significant confusion or complaints, we will stop distributing the man page. The alternative, updating the man page when we update the Info file, is impossible because the rest of the work of maintaining GNU CC leaves us no time for that. The GNU project regards man pages as obsolete and should not let them take time away from other things.

For complete and current documentation, refer to the Info file '**gcc'** or the manual *Using and Porting GNU CC (for version 2.0)*. Both are made from the Texinfo source file **gcc.texinfo**.

DESCRIPTION (edited for use in CS 9C and 9F)

The C and C++ compilers are integrated. Both process input files through one or more of four stages: preprocessing, compilation, assembly, and linking. Source filename suffixes identify the source language, but which name you use for the compiler governs default assumptions:

gcc assumes preprocessed (**.i**) files are C and assumes C style linking.

g++ assumes preprocessed (**.i**) files are C++ and assumes C++ style linking.

Suffixes of source file names indicate the language and kind of processing to be done:

.c C source; preprocess, compile, assemble
.C C++ source; preprocess, compile, assemble
.cc C++ source; preprocess, compile, assemble
.cxx C++ source; preprocess, compile, assemble
.m Objective-C source; preprocess, compile, assemble
.i preprocessed C; compile, assemble
.ii preprocessed C++; compile, assemble
.s Assembler source; assemble
.S Assembler source; preprocess, assemble
.h Preprocessor file; not usually named on command line

Files with other suffixes are passed to the linker. Common cases include:

.o Object file
.a Archive file

Linking is always the last stage unless you use one of the **–c**, **–S**, or **–E** options to avoid it (or unless compilation errors stop the whole process). For the link stage, all **.o** files corresponding to source files, **–l** libraries, unrecognized filenames (including named **.o** object files and **.a** archives) are passed to the linker in command-line order.

OPTIONS

Options must be separate: '**–dr'** is quite different from **'–d –r** '.

Most '**–f**' and '**–W**' options have two contrary forms: **–f***name* and **–fno–***name* (or **–W***name* and **–Wno–***name)*.Only the non-default forms are shown here.

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

–c –S –E –o*file* –pipe –v –x*language*

Language Options

–ansi –fall–virtual –fcond–mismatch –fdollars–in–identifiers –fenum–int–equiv
–fexternal-templates –fno–asm –fno–builtin –fno–strict–prototype –fsigned–bitfields –fsigned–char
–fthis–is–variable –funsigned–bitfields –funsigned–char –fwritable–strings –traditional
–traditional–cpp –trigraphs

Warning Options

–fsyntax–only –pedantic –pedantic–errors –w –W –Wall –Waggregate–return
–Wcast–align –Wcast–qual –Wchar–subscript –Wcomment –Wconversion –Wenum–clash
–Werror –Wformat –Wid–clash–*len* –Wimplicit –Winline –Wmissing–prototypes
–Wmissing–declarations –Wnested–externs –Wno–import –Wparentheses –Wpointer–arith
–Wredundant–decls –Wreturn–type –Wshadow –Wstrict–prototypes –Wswitch
–Wtemplate–debugging –Wtraditional –Wtrigraphs –Wuninitialized –Wunused –Wwrite–strings

Debugging Options (edited for use in CS 9C and 9F)

–g –g*level* –p –pg

Optimization Options (edited for use in CS 9C and 9F)

–O –O2

Preprocessor Options (edited for use in CS 9C and 9F)

–include *file* –M –MD –MM –MMD

Assembler Option

–Wa,*option*

Linker Options

–l*library* –nostartfiles –nostdlib –static –shared –symbolic –Xlinker *option* –Wl,*option* –u *symbol*

Directory Options

–B*prefix* –I*dir* –I– –L*dir*

Target Options (see online man entry for information)

Configuration Dependent Options (see online man entry for information)

Code Generation Options (see online man entry for information)

OVERALL OPTIONS

–x*language*

Specify explicitly the *language* for the following input files (rather than choosing a default based on the file name suffix) . 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).

If you want only some of the four stages (preprocess, compile, assemble, link), 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 compiler output is an object file corresponding to each source file.

By default, GCC makes the object file name for a source file by replacing the suffix '**.c', '.i', '.s**', etc., with '**.o**'. Use **–o** to select another name.

GCC ignores any unrecognized input files (those that do not require compilation or assembly) with the **–c** option.

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

By default, GCC makes the assembler file name for a source file by replacing the suffix '**.c', '.i**', etc., with '**.s**'. Use **–o** to select another name.

GCC ignores any input files that don't require compilation.

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

GCC ignores input files which don't require preprocessing.

–o *file*

Place output in file *file*. This applies regardless to whatever sort of output GCC is producing, 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 you do not specify '**–o'**, 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 Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler cannot read from a pipe; but the GNU assembler has no trouble.

LANGUAGE OPTIONS

The following options control the dialect of 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, and disallows '**$'** as part of identifiers.

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'.**

The preprocessor predefines a macro **__STRICT_ANSI__** when you use the **'–ansi'** option. 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.

–fno–asm

Do not 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'.**

–fno–builtin

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

The '**–ansi'** option prevents **alloca** and **_exit** from being builtin functions.

–fno–strict–prototype

Treat a function declaration with no arguments, such as '**int foo** ( );', as C would treat it—as saying nothing about the number of arguments or their types (C++ only). Normally, such a declaration in C++ means that the function **foo** takes no arguments.

–trigraphs

Support ANSI C trigraphs. The '**–ansi'** option implies **'–trigraphs'.**

–traditional

Attempt to support some aspects of traditional C compilers. For details, see the GNU C Manual; the duplicate list here has been deleted so that we won't get complaints when it is out of date.

But one note about C++ programs only (not C). '**–traditional'** has one additional effect for C++: assignment to **this** is permitted. This is the same as the effect of '**–fthis–is–variable'.**

–traditional–cpp

Attempt to support some aspects of traditional C preprocessors. This includes the items that specifically mention the preprocessor above, but none of the other effects of '**–traditional'.**

–fdollars–in–identifiers

Permit the use of '**$'** in identifiers (C++ only). You can also use '**–fno–dollars–in–identifiers'** to explicitly prohibit use of '**$'**. (GNU C++ allows **'$'** by default on some target systems but not others.)

–fenum–int–equiv

Permit implicit conversion of **int** to enumeration types (C++ only). Normally GNU C++ allows conversion of **enum to int,** but not the other way around.

–fexternal–templates

Produce smaller code for template declarations, by generating only a single copy of each template function where it is defined (C++ only). To use this option successfully, you must also mark all files that use templates with either '**#pragma implementation'** (the definition) or '**#pragma interface'** (declarations).

When your code is compiled with '**–fexternal–templates'**, all template instantiations are external. You must arrange for all necessary instantiations to appear in the implementation file; you can do this with a **typedef** that references each instantiation needed. Conversely, when you compile using the default option '**–fno–external–templates'**, all template instantiations are explicitly internal.

–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.

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

–fthis–is–variable
Permit assignment to **this** (C++ only). 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. However, for backwards compatibility, you can make it valid with '**–fthis-is-variable'.**

–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** and **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, '**–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 declared with no explicit '**signed'** or **'unsigned'** qualifier. By default, such a bitfield is signed, because this is consistent: the basic integer types such as **int** are signed types.

However, when you specify '**–traditional'**, 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.

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

PREPROCESSOR OPTIONS (edited for use in CS 9C and 9F)

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

–include*file*
Process *file* as input before processing the regular input file. In effect, the contents of *file* are compiled first. All the '**–include'** options are processed in the order in which they are written.

–M[–MG]

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 '****'-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

'**–M'** implies **'–E'.**

'**–MG'** says to treat missing header files as generated files and assume they live in the same directory as the source file. It must be specified in addition to **'–M'.**

–MM [–MG]

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 '**.o'** with '**.d**' at the end of the output file names. This is in addition to compiling the file as specified—'**–MD'** does not inhibit ordinary compilation the way '**–M'** does.

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

ASSEMBLER OPTION (see online man entry for information)

LINKER OPTIONS

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 GCC does a link step, these object files are used as input to the linker.

–l*library*

Use the library named *library* when linking.

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. However, if the linker finds an ordinary object file rather than a library, the object file 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.

–lobjc

You need this special case of the **–l** option in order to link an Objective C program.

–nostartfiles

Do not use the standard system startup files when linking. The standard libraries are used normally.

–nostdlib

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

–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.

–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.

DIRECTORY OPTIONS

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

–I*dir* Append directory *dir* to the list of directories searched for include files.

–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'.**

–B*prefix*
This option specifies where to find the executables, libraries and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms '**cpp', 'cc1'** (or, for **C++, 'cc1plus'), 'as'** and **'ld'**. It tries *prefix* as a prefix for each program it tries to run, both with and without '*machine*/*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 compiler driver searches for the unmodified program name, using the directories specified in your '**PATH'** environment variable.

The run-time support file '**libgcc.a'** is also 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. Most of the time, on most machines, '**libgcc.a'** is not actually necessary.

You can get a similar result from the environment variable **GCC_EXEC_PREFIX**; if it is defined, its value is used as a prefix in the same way. If both the '**–B'** option and the **GCC_EXEC_PREFIX** variable are present, the **'–B'** option is used first and the environment variable value second.

WARNING OPTIONS

Warnings are diagnostic messages that report constructions which are not 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:

–fsyntax–only
Check the code for syntax errors, but don't emit any output.

–w Inhibit all warning messages.

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

–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. There is no reason to *use* this option; it exists only to satisfy pedants.

'**–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.

–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 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;
}
```

Spurious warnings can occur because GNU CC does not 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 is not **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** and **scanf**, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

–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.

–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 as **volatile** all the functions you use that never return.

–Wparentheses
Warn if parentheses are omitted in certain contexts.

–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.

–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.

–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- 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.

–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.

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

–Woverloaded–virtual
(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. Use this option to request warnings when a derived class declares a function that may be an erroneous attempt to define a virtual function: that is, warn when a function with the same name as a virtual function in the base class, but with a type signature that doesn’t match any virtual functions from the base class.

–Winline

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

–Werror

Treat warnings as errors; abort compilation after any warning.

DEBUGGING OPTIONS (edited for use in CS 9C and 9F)

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 (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

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.

–g*level* 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 Generate extra code to write profile information suitable for the analysis program **prof.**

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

OPTIMIZATION OPTIONS (edited for use in CS 9C and 9F)

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'**, only variables declared **register** are allocated 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.

–O2 Optimize even more. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example. As compared to **–O,** this option increases both compilation time and the performance of the generated code.

–O3 Optimize yet more.

–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.

TARGET OPTIONS (see online man entry for information)

MACHINE DEPENDENT OPTIONS (see online man entry for information)

CODE GENERATION OPTIONS (see online man entry for information)

PRAGMAS

Two '**#pragma'** directives are supported for GNU C++, to permit using the same header file for two purposes: as a definition of interfaces to a given object class, and as the full definition of the contents of that object class.

#pragma interface

(C++ only.) 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.

#pragma implementation

#pragma implementation "*objects***.h"**

(C++ only.) 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; for example, in '**allclass.cc', '#pragma implementation'** by itself is equivalent to **'#pragma implementation "allclass.h"'.** Use the string argument if you want a single implementation file to include code from multiple header files.

There is no way to split up the contents of a single header file into multiple implementation files.

FILES

file.c	C source file
file.h	C header (preprocessor) file
file.i	preprocessed C source file
file.C	C++ source file
file.cc	C++ source file
file.cxx	C++ source file
file.m	Objective-C source file
file.s	assembly language file
file.o	object file
a.out	link edited output
TMPDIR/cc*	temporary files
LIBDIR/cpp	preprocessor
LIBDIR/cc1	compiler for C
LIBDIR/cc1plus	compiler for C++
LIBDIR/collect	linker front end needed on some machines
LIBDIR/libgcc.a	GCC subroutine library

/lib/crt[01n].o	start-up routine
LIBDIR/ccrt0	additional start-up routine for C++
/lib/libc.a	standard C library, see *intro*(3)
/usr/include	standard directory for **#include** files
LIBDIR/include	standard gcc directory for **#include** files
LIBDIR/g++–include	additional g++ directory for **#include**

LIBDIR is usually **/usr/local/lib/***machine*/*version*.
TMPDIR comes from the environment variable **TMPDIR** (default **/usr/tmp** if available, else **/tmp).**

SEE ALSO

cpp(1), as(1), ld(1), gdb(1), adb(1), dbx(1), sdb(1).
'**gcc**','**cpp**', '**as**','**ld**', and '**gdb**' entries in **info.**
Using and Porting GNU CC (for version 2.0), Richard M. Stallman; *The C Preprocessor*, Richard M. Stallman; *Debugging with GDB: the GNU Source-Level Debugger*, Richard M. Stallman and Roland H. Pesch; *Using as: the GNU Assembler,* Dean Elsner, Jay Fenlason & friends; *ld: the GNU linker*, Steve Chamberlain and Roland Pesch.

BUGS

For instructions on reporting bugs, see the GCC manual.

COPYING

AUTHORS

See the GNU CC Manual for the contributors to GNU CC.