Using the GNU Compiler Collection

Richard M. Stallman

Last updated 20 April 2002

for GCC 3.1

For GCC Version 3.1

Published by the Free Software Foundation
59 Temple Place—Suite 330
Boston, MA 02111-1307, USA
Last printed April, 1998.
Printed copies are available for $50 each.

Short Contents

Table of Contents

Introduction

This manual documents how to use the GNU compilers, as well as their features and incompatibilities, and how to report bugs. It corresponds to GCC version 3.1. The internals of the GNU compilers, including how to port them to new targets and some information about how to write front ends for new languages, are documented in a separate manual. See section "Introduction" in *GNU Compiler Collection (GCC) Internals*.

1 Compile C, C++, Objective-C, Ada, Fortran, or Java

Several versions of the compiler (C, C++, Objective-C, Ada, Fortran, and Java) are integrated; this is why we use the name "GNU Compiler Collection". GCC can compile programs written in any of these languages. The Ada, Fortran, and Java compilers are described in separate manuals.

"GCC" is a common shorthand term for the GNU Compiler Collection. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs (as the abbreviation formerly stood for "GNU C Compiler").

When referring to C++ compilation, it is usual to call the compiler "G++". Since there is only one compiler, it is also accurate to call it "GCC" no matter what the language context; however, the term "G++" is more useful when the emphasis is on compiling C++ programs.

Similarly, when we talk about Ada compilation, we usually call the compiler "GNAT", for the same reasons.

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

Front ends for other languages, such as Mercury and Pascal exist but have not yet been integrated into GCC. These front ends, like that for C++, are built in subdirectories of GCC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective-C, or any of the languages for which you have installed front ends.

In this manual, we only discuss the options for the C, Objective-C, and C++ compilers and those of the GCC core. Consult the documentation of the other front ends for the options to use when compiling programs written in other languages.

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

2 Language Standards Supported by GCC

For each language compiled by GCC for which there is a standard, GCC attempts to follow one or more versions of that standard, possibly with some exceptions, and possibly with some extensions.

GCC supports three versions of the C standard, although support for the most recent version is not yet complete.

The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990. This standard was ratified as an ISO standard (ISO/IEC 9899:1990) later in 1990. There were no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. This standard, in both its forms, is commonly known as *C89*, or occasionally as *C90*, from the dates of ratification. The ANSI standard, but not the ISO standard, also came with a Rationale document. To select this standard in GCC, use one of the options '`-ansi`', '`-std=c89`' or '`-std=iso9899:1990`'; to obtain all the diagnostics required by the standard, you should also specify '`-pedantic`' (or '`-pedantic-errors`' if you want them to be errors rather than warnings). See Section 3.4 [Options Controlling C Dialect], page 19.

Errors in the 1990 ISO C standard were corrected in two Technical Corrigenda published in 1994 and 1996. GCC does not support the uncorrected version.

An amendment to the 1990 standard was published in 1995. This amendment added digraphs and `__STDC_VERSION__` to the language, but otherwise concerned the library. This amendment is commonly known as *AMD1*; the amended standard is sometimes known as *C94* or *C95*. To select this standard in GCC, use the option '`-std=iso9899:199409`' (with, as for other standard versions, '`-pedantic`' to receive all required diagnostics).

A new edition of the ISO C standard was published in 1999 as ISO/IEC 9899:1999, and is commonly known as *C99*. GCC has incomplete support for this standard version; see `http://gcc.gnu.org/gcc-3.1/c99status.html` for details. To select this standard, use '`-std=c99`' or '`-std=iso9899:1999`'. (While in development, drafts of this standard version were referred to as *C9X*.)

Errors in the 1999 ISO C standard were corrected in a Technical Corrigendum published in 2001. GCC does not support the uncorrected version.

GCC also has some limited support for traditional (pre-ISO) C with the '`-traditional`' option. This support may be of use for compiling some very old programs that have not been updated to ISO C, but should not be used for new programs. It will not work with some modern C libraries such as the GNU C library.

By default, GCC provides some extensions to the C language that on rare occasions conflict with the C standard. See Chapter 5 [Extensions to the C Language Family], page 155. Use of the '`-std`' options listed above will disable these extensions where they conflict with the C standard version selected. You may also select an extended version of the C language explicitly with '`-std=gnu89`' (for C89 with GNU extensions) or '`-std=gnu99`' (for C99 with GNU extensions). The default, if no C language dialect options are given, is '`-std=gnu89`'; this will change to '`-std=gnu99`' in some future release when the C99 support is complete. Some features that are part of the C99 standard are accepted as extensions in C89 mode.

The ISO C standard defines (in clause 4) two classes of conforming implementation. A *conforming hosted implementation* supports the whole standard including all the library facilities; a *conforming freestanding implementation* is only required to provide certain library facilities: those in `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`; since AMD1, also those in `<iso646.h>`; and in C99, also those in `<stdbool.h>` and `<stdint.h>`. In addition, complex types, added in C99, are not required for freestanding implementations. The standard also defines two environments for programs, a *freestanding environment*, required of all implementations and which may not have library facilities beyond those required of freestanding implementations, where the handling of program startup and termination are implementation-defined, and a *hosted environment*, which is not required, in which all the library facilities are provided and startup is through a function `int main (void)` or `int main (int, char *[])`. An OS kernel would be a freestanding environment; a program using the facilities of an operating system would normally be in a hosted implementation.

GCC aims towards being usable as a conforming freestanding implementation, or as the compiler for a conforming hosted implementation. By default, it will act as the compiler for a hosted implementation, defining `__STDC_HOSTED__` as `1` and presuming that when the names of ISO C functions are used, they have the semantics defined in the standard. To make it act as a conforming freestanding implementation for a freestanding environment, use the option '`-ffreestanding`'; it will then define `__STDC_HOSTED__` to `0` and not make assumptions about the meanings of function names from the standard library, with exceptions noted below. To build an OS kernel, you may well still need to make your own arrangements for linking and startup. See Section 3.4 [Options Controlling C Dialect], page 19.

GCC does not provide the library facilities required only of hosted implementations, nor yet all the facilities required by C99 of freestanding implementations; to use the facilities of a hosted environment, you will need to find them elsewhere (for example, in the GNU C library). See Section 9.7 [Standard Libraries], page 282.

Most of the compiler support routines used by GCC are present in '`libgcc`', but there are a few exceptions. GCC requires the freestanding environment provide `memcpy`, `memmove`, `memset` and `memcmp`. Some older ports of GCC are configured to use the BSD `bcopy`, `bzero` and `bcmp` functions instead, but this is deprecated for new ports. Finally, if `__builtin_trap` is used, and the target does not implement the `trap` pattern, then GCC will emit a call to `abort`.

For references to Technical Corrigenda, Rationale documents and information concerning the history of C that is available online, see `http://gcc.gnu.org/readings.html`

There is no formal written standard for Objective-C. The most authoritative manual is "Object-Oriented Programming and the Objective-C Language", available at a number of web sites; `http://developer.apple.com/techpubs/macosx/Cocoa/ObjectiveC/` has a recent version, while `http://www.toodarkpark.org/computers/objc/` is an older example. `http://www.gnustep.org` includes useful information as well.

See section "About This Guide" in *GNAT Reference Manual*, for information on standard conformance and compatibility of the Ada compiler.

See section "The GNU Fortran Language" in *Using and Porting GNU Fortran*, for details of the Fortran language supported by GCC.

See section "Compatibility with the Java Platform" in *GNU gcj*, for details of compatibility between `gcj` and the Java Platform.

3 GCC Command Options

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

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

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

See Section 3.3 [Compiling C++ Programs], page 19, for a summary of special options for compiling C++ programs.

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

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

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

See [Option Index], page 337, for an index to GCC's options.

3.1 Option Summary

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

Overall Options

See Section 3.2 [Options Controlling the Kind of Output], page 16.

```
-c -S -E -o file -pipe -pass-exit-codes -x language
-v -### --target-help --help
```

C Language Options

See Section 3.4 [Options Controlling C Dialect], page 19.

```
-ansi -std=standard -aux-info filename
-fno-asm -fno-builtin -fno-builtin-function
-fhosted -ffreestanding
-trigraphs -no-integrated-cpp -traditional -traditional-cpp
-fallow-single-precision -fcond-mismatch
-fsigned-bitfields -fsigned-char
-funsigned-bitfields -funsigned-char
-fwritable-strings -fshort-wchar
```

C++ Language Options

See Section 3.5 [Options Controlling C++ Dialect], page 24.

```
-fno-access-control -fcheck-new -fconserve-space
-fno-const-strings -fdollars-in-identifiers
-fno-elide-constructors
-fno-enforce-eh-specs -fexternal-templates
-falt-external-templates
-ffor-scope -fno-for-scope -fno-gnu-keywords
-fno-implicit-templates
-fno-implicit-inline-templates
-fno-implement-inlines -fms-extensions
-fno-nonansi-builtins -fno-operator-names
-fno-optional-diags -fpermissive
-frepo -fno-rtti -fstats -ftemplate-depth-n
-fuse-cxa-atexit -fvtable-gc -fno-weak -nostdinc++
-fno-default-inline -Wctor-dtor-privacy
-Wnon-virtual-dtor -Wreorder
-Weffc++ -Wno-deprecated
-Wno-non-template-friend -Wold-style-cast
-Woverloaded-virtual -Wno-pmf-conversions
-Wsign-promo -Wsynth
```

Objective-C Language Options

See Section 3.6 [Options Controlling Objective-C Dialect], page 30.

```
-fconstant-string-class=class-name
-fgnu-runtime -fnext-runtime -gen-decls
-Wno-protocol -Wselector
```

Language Independent Options

See Section 3.7 [Options to Control Diagnostic Messages Formatting], page 31.

```
-fmessage-length=n
-fdiagnostics-show-location=[once|every-line]
```

Warning Options

See Section 3.8 [Options to Request or Suppress Warnings], page 31.

```
-fsyntax-only -pedantic -pedantic-errors
-w -W -Wall -Waggregate-return
-Wcast-align -Wcast-qual -Wchar-subscripts -Wcomment
-Wconversion -Wno-deprecated-declarations
-Wdisabled-optimization -Wdiv-by-zero -Werror
-Wfloat-equal -Wformat -Wformat=2
-Wformat-nonliteral -Wformat-security
-Wimplicit -Wimplicit-int
-Wimplicit-function-declaration
-Werror-implicit-function-declaration
-Wimport -Winline
-Wlarger-than-len -Wlong-long
-Wmain -Wmissing-braces -Wmissing-declarations
-Wmissing-format-attribute -Wmissing-noreturn
-Wmultichar -Wno-format-extra-args -Wno-format-y2k
```

```
-Wno-import -Wpacked -Wpadded
-Wparentheses -Wpointer-arith -Wredundant-decls
-Wreturn-type -Wsequence-point -Wshadow
-Wsign-compare -Wswitch -Wsystem-headers
-Wtrigraphs -Wundef -Wuninitialized
-Wunknown-pragmas -Wunreachable-code
-Wunused -Wunused-function -Wunused-label -Wunused-parameter
-Wunused-value -Wunused-variable -Wwrite-strings
```

C-only Warning Options

```
-Wbad-function-cast -Wmissing-prototypes -Wnested-externs
-Wstrict-prototypes -Wtraditional
```

Debugging Options

See Section 3.9 [Options for Debugging Your Program or GCC], page 43.

```
-dletters -dumpspecs -dumpmachine -dumpversion
-fdump-unnumbered -fdump-translation-unit[-n]
-fdump-class-hierarchy[-n]
-fdump-tree-original[-n] -fdump-tree-optimized[-n]
-fdump-tree-inlined[-n]
-fmem-report -fpretend-float
-fprofile-arcs -ftest-coverage -ftime-report
-g -glevel -gcoff -gdwarf -gdwarf-1 -gdwarf-1+ -gdwarf-2
-ggdb -gstabs -gstabs+ -gvms -gxcoff -gxcoff+
-p -pg -print-file-name=library -print-libgcc-file-name
-print-multi-directory -print-multi-lib
-print-prog-name=program -print-search-dirs -Q
-save-temps -time
```

Optimization Options

See Section 3.10 [Options that Control Optimization], page 50.

```
-falign-functions=n -falign-jumps=n
-falign-labels=n -falign-loops=n
-fbranch-probabilities -fcaller-saves -fcprop-registers
-fcse-follow-jumps -fcse-skip-blocks -fdata-sections
-fdelayed-branch -fdelete-null-pointer-checks
-fexpensive-optimizations -ffast-math -ffloat-store
-fforce-addr -fforce-mem -ffunction-sections
-fgcse -fgcse-lm -fgcse-sm
-finline-functions -finline-limit=n -fkeep-inline-functions
-fkeep-static-consts -fmerge-constants -fmerge-all-constants
-fmove-all-movables -fno-default-inline -fno-defer-pop
-fno-function-cse -fno-guess-branch-probability
-fno-inline -fno-math-errno -fno-peephole -fno-peephole2
-funsafe-math-optimizations -fno-trapping-math
-fomit-frame-pointer -foptimize-register-move
-foptimize-sibling-calls -fprefetch-loop-arrays
-freduce-all-givs -fregmove -frename-registers
-frerun-cse-after-loop -frerun-loop-opt
-fschedule-insns -fschedule-insns2
-fsingle-precision-constant -fssa -fssa-ccp -fssa-dce
```

-fstrength-reduce -fstrict-aliasing -fthread-jumps -ftrapv
-funroll-all-loops -funroll-loops
--param *name*=*value* -O -O0 -O1 -O2 -O3 -Os

Preprocessor Options

See Section 3.11 [Options Controlling the Preprocessor], page 61.

-$ -A*question*=*answer* -A-*question*[=*answer*]
-C -dD -dI -dM -dN
-D*macro*[=*defn*] -E -H
-idirafter *dir*
-include *file* -imacros *file*
-iprefix *file* -iwithprefix *dir*
-iwithprefixbefore *dir* -isystem *dir*
-M -MM -MF -MG -MP -MQ -MT -nostdinc -P -remap
-trigraphs -undef -U*macro* -Wp,*option*

Assembler Option

See Section 3.12 [Passing Options to the Assembler], page 68.

-Wa,*option*

Linker Options

See Section 3.13 [Options for Linking], page 68.

object-file-name -l*library*
-nostartfiles -nodefaultlibs -nostdlib
-s -static -static-libgcc -shared -shared-libgcc -symbolic
-Wl,*option* -Xlinker *option*
-u *symbol*

Directory Options

See Section 3.14 [Options for Directory Search], page 71.

-B*prefix* -I*dir* -I- -L*dir* -specs=*file*

Target Options

See Section 3.16 [Target Options], page 79.

-b *machine* -V *version*

Machine Dependent Options

See Section 3.17 [Hardware Models and Configurations], page 80.

M680x0 Options

-m68000 -m68020 -m68020-40 -m68020-60 -m68030 -m68040
-m68060 -mcpu32 -m5200 -m68881 -mbitfield -mc68000 -mc68020
-mfpa -mnobitfield -mrtd -mshort -msoft-float -mpcrel
-malign-int -mstrict-align

M68hc1x Options

-m6811 -m6812 -m68hc11 -m68hc12
-mauto-incdec -mshort -msoft-reg-count=*count*

VAX Options

-mg -mgnu -munix

SPARC Options

```
-mcpu=cpu-type
-mtune=cpu-type
-mcmodel=code-model
-m32 -m64
-mapp-regs -mbroken-saverestore -mcypress
-mfaster-structs -mflat
-mfpu -mhard-float -mhard-quad-float
-mimpure-text -mlive-g0 -mno-app-regs
-mno-faster-structs -mno-flat -mno-fpu
-mno-impure-text -mno-stack-bias -mno-unaligned-doubles
-msoft-float -msoft-quad-float -msparclite -mstack-bias
-msupersparc -munaligned-doubles -mv8
```

Convex Options

```
-mc1 -mc2 -mc32 -mc34 -mc38
-margcount -mnoargcount
-mlong32 -mlong64
-mvolatile-cache -mvolatile-nocache
```

AMD29K Options

```
-m29000 -m29050 -mbw -mnbw -mdw -mndw
-mlarge -mnormal -msmall
-mkernel-registers -mno-reuse-arg-regs
-mno-stack-check -mno-storem-bug
-mreuse-arg-regs -msoft-float -mstack-check
-mstorem-bug -muser-registers
```

ARM Options

```
-mapcs-frame -mno-apcs-frame
-mapcs-26 -mapcs-32
-mapcs-stack-check -mno-apcs-stack-check
-mapcs-float -mno-apcs-float
-mapcs-reentrant -mno-apcs-reentrant
-msched-prolog -mno-sched-prolog
-mlittle-endian -mbig-endian -mwords-little-endian
-malignment-traps -mno-alignment-traps
-msoft-float -mhard-float -mfpe
-mthumb-interwork -mno-thumb-interwork
-mcpu=name -march=name -mfpe=name
-mstructure-size-boundary=n
-mbsd -mxopen -mno-symrename
-mabort-on-noreturn
-mlong-calls -mno-long-calls
-msingle-pic-base -mno-single-pic-base
-mpic-register=reg
-mnop-fun-dllimport
-mpoke-function-name
-mthumb -marm
-mtpcs-frame -mtpcs-leaf-frame
-mcaller-super-interworking -mcallee-super-interworking
```

MN10200 Options

```
-mrelax
```

MN10300 Options

```
-mmult-bug -mno-mult-bug
-mam33 -mno-am33
-mno-crt0 -mrelax
```

M32R/D Options

```
-m32rx -m32r -mcode-model=model-type -msdata=sdata-type
-G num
```

M88K Options

```
-m88000 -m88100 -m88110 -mbig-pic
-mcheck-zero-division -mhandle-large-shift
-midentify-revision -mno-check-zero-division
-mno-ocs-debug-info -mno-ocs-frame-position
-mno-optimize-arg-area -mno-serialize-volatile
-mno-underscores -mocs-debug-info
-mocs-frame-position -moptimize-arg-area
-mserialize-volatile -mshort-data-num -msvr3
-msvr4 -mtrap-large-shift -muse-div-instruction
-mversion-03.00 -mwarn-passed-structs
```

RS/6000 and PowerPC Options

```
-mcpu=cpu-type
-mtune=cpu-type
-mpower -mno-power -mpower2 -mno-power2
-mpowerpc -mpowerpc64 -mno-powerpc
-maltivec -mno-altivec
-mpowerpc-gpopt -mno-powerpc-gpopt
-mpowerpc-gfxopt -mno-powerpc-gfxopt
-mnew-mnemonics -mold-mnemonics
-mfull-toc -mminimal-toc -mno-fp-in-toc -mno-sum-in-toc
-m64 -m32 -mxl-call -mno-xl-call -mpe
-msoft-float -mhard-float -mmultiple -mno-multiple
-mstring -mno-string -mupdate -mno-update
-mfused-madd -mno-fused-madd -mbit-align -mno-bit-align
-mstrict-align -mno-strict-align -mrelocatable
-mno-relocatable -mrelocatable-lib -mno-relocatable-lib
-mtoc -mno-toc -mlittle -mlittle-endian -mbig -mbig-endian
-mcall-aix -mcall-sysv -mcall-netbsd
-maix-struct-return -msvr4-struct-return -mabi=altivec -mabi=no-altivec
-mprototype -mno-prototype
-msim -mmvme -mads -myellowknife -memb -msdata
-msdata=opt -mvxworks -G num -pthread
```

RT Options

```
-mcall-lib-mul -mfp-arg-in-fpregs -mfp-arg-in-gregs
-mfull-fp-blocks -mhc-struct-return -min-line-mul
-mminimum-fp-blocks -mnohc-struct-return
```

MIPS Options

```
-mabicalls -march=cpu-type -mtune=cpu=type
-mcpu=cpu-type -membedded-data -muninit-const-in-rodata
-membedded-pic -mfp32 -mfp64 -mfused-madd -mno-fused-madd
-mgas -mgp32 -mgp64
-mgpopt -mhalf-pic -mhard-float -mint64 -mips1
-mips2 -mips3 -mips4 -mlong64 -mlong32 -mlong-calls -mmemcpy
-mmips-as -mmips-tfile -mno-abicalls
-mno-embedded-data -mno-uninit-const-in-rodata
-mno-embedded-pic -mno-gpopt -mno-long-calls
-mno-memcpy -mno-mips-tfile -mno-rnames -mno-stats
-mrnames -msoft-float
-m4650 -msingle-float -mmad
-mstats -EL -EB -G num -nocpp
-mabi=32 -mabi=n32 -mabi=64 -mabi=eabi
-mfix7000 -mno-crt0 -mflush-func=func -mno-flush-func
```

i386 and x86-64 Options

```
-mcpu=cpu-type -march=cpu-type -mfpmath=unit
-masm=dialect -mno-fancy-math-387
-mno-fp-ret-in-387 -msoft-float -msvr3-shlib
-mno-wide-multiply -mrtd -malign-double
-mpreferred-stack-boundary=num
-mmmx -msse -msse2 -msse-math -m3dnow
-mthreads -mno-align-stringops -minline-all-stringops
-mpush-args -maccumulate-outgoing-args -m128bit-long-double
-m96bit-long-double -mregparm=num -momit-leaf-frame-pointer
-mno-red-zone
-m32 -m64
```

HPPA Options

```
-march=architecture-type
-mbig-switch -mdisable-fpregs -mdisable-indexing
-mfast-indirect-calls -mgas -mjump-in-delay
-mlong-load-store -mno-big-switch -mno-disable-fpregs
-mno-disable-indexing -mno-fast-indirect-calls -mno-gas
-mno-jump-in-delay -mno-long-load-store
-mno-portable-runtime -mno-soft-float
-mno-space-regs -msoft-float -mpa-risc-1-0
-mpa-risc-1-1 -mpa-risc-2-0 -mportable-runtime
-mschedule=cpu-type -mspace-regs
```

Intel 960 Options

```
-mcpu-type -masm-compat -mclean-linkage
-mcode-align -mcomplex-addr -mleaf-procedures
-mic-compat -mic2.0-compat -mic3.0-compat
-mintel-asm -mno-clean-linkage -mno-code-align
-mno-complex-addr -mno-leaf-procedures
-mno-old-align -mno-strict-align -mno-tail-call
-mnumerics -mold-align -msoft-float -mstrict-align
-mtail-call
```

DEC Alpha Options

-mno-fp-regs -msoft-float -malpha-as -mgas
-mieee -mieee-with-inexact -mieee-conformant
-mfp-trap-mode=*mode* -mfp-rounding-mode=*mode*
-mtrap-precision=*mode* -mbuild-constants
-mcpu=*cpu-type* -mtune=*cpu-type*
-mbwx -mmax -mfix -mcix
-mfloat-vax -mfloat-ieee
-mexplicit-relocs -msmall-data -mlarge-data
-mmemory-latency=*time*

DEC Alpha/VMS Options

-mvms-return-codes

Clipper Options

-mc300 -mc400

H8/300 Options

-mrelax -mh -ms -mint32 -malign-300

SH Options

-m1 -m2 -m3 -m3e
-m4-nofpu -m4-single-only -m4-single -m4
-m5-64media -m5-64media-nofpu
-m5-32media -m5-32media-nofpu
-m5-compact -m5-compact-nofpu
-mb -ml -mdalign -mrelax
-mbigtable -mfmovd -mhitachi -mnomacsave
-mieee -misize -mpadstruct -mspace
-mprefergot -musermode

System V Options

-Qy -Qn -YP,*paths* -Ym,*dir*

ARC Options

-EB -EL
-mmangle-cpu -mcpu=*cpu* -mtext=*text-section*
-mdata=*data-section* -mrodata=*readonly-data-section*

TMS320C3x/C4x Options

-mcpu=*cpu* -mbig -msmall -mregparm -mmemparm
-mfast-fix -mmpyi -mbk -mti -mdp-isr-reload
-mrpts=*count* -mrptb -mdb -mloop-unsigned
-mparallel-insns -mparallel-mpy -mpreserve-float

V850 Options

-mlong-calls -mno-long-calls -mep -mno-ep
-mprolog-function -mno-prolog-function -mspace
-mtda=*n* -msda=*n* -mzda=*n*
-mv850 -mbig-switch

NS32K Options

-m32032 -m32332 -m32532 -m32081 -m32381
-mmult-add -mnomult-add -msoft-float -mrtd -mnortd

```
-mregparam -mnoregparam -msb -mnosb
-mbitfield -mnobitfield -mhimem -mnohimem
```

AVR Options

```
-mmcu=mcu -msize -minit-stack=n -mno-interrupts
-mcall-prologues -mno-tablejump -mtiny-stack
```

MCore Options

```
-mhardlit -mno-hardlit -mdiv -mno-div -mrelax-immediates
-mno-relax-immediates -mwide-bitfields -mno-wide-bitfields
-m4byte-functions -mno-4byte-functions -mcallgraph-data
-mno-callgraph-data -mslow-bytes -mno-slow-bytes -mno-lsim
-mlittle-endian -mbig-endian -m210 -m340 -mstack-increment
```

MMIX Options

```
-mlibfuncs -mno-libfuncs -mepsilon -mno-epsilon -mabi=gnu
-mabi=mmixware -mzero-extend -mknuthdiv -mtoplevel-symbols
-melf -mbranch-predict -mno-branch-predict -mbase-addresses
-mno-base-addresses
```

IA-64 Options

```
-mbig-endian -mlittle-endian -mgnu-as -mgnu-ld -mno-pic
-mvolatile-asm-stop -mb-step -mregister-names -mno-sdata
-mconstant-gp -mauto-pic -minline-divide-min-latency
-minline-divide-max-throughput -mno-dwarf2-asm
-mfixed-range=register-range
```

D30V Options

```
-mextmem -mextmemory -monchip -mno-asm-optimize -masm-optimize
-mbranch-cost=n -mcond-exec=n
```

S/390 and zSeries Options

```
-mhard-float -msoft-float -mbackchain -mno-backchain
-msmall-exec -mno-small-exec -mmvcle -mno-mvcle
-m64 -m31 -mdebug -mno-debug
```

CRIS Options

```
-mcpu=cpu -march=cpu -mtune=cpu
-mmax-stack-frame=n -melinux-stacksize=n
-metrax4 -metrax100 -mpdebug -mcc-init -mno-side-effects
-mstack-align -mdata-align -mconst-align
-m32-bit -m16-bit -m8-bit -mno-prologue-epilogue -mno-gotplt
-melf -maout -melinux -mlinux -sim -sim2
```

PDP-11 Options

```
-mfpu -msoft-float -mac0 -mno-ac0 -m40 -m45 -m10
-mbcopy -mbcopy-builtin -mint32 -mno-int16
-mint16 -mno-int32 -mfloat32 -mno-float64
-mfloat64 -mno-float32 -mabshi -mno-abshi
-mbranch-expensive -mbranch-cheap
-msplit -mno-split -munix-asm -mdec-asm
```

Xstormy16 Options

```
-msim
```

Xtensa Options

```
-mbig-endian -mlittle-endian
-mdensity -mno-density
-mmac16 -mno-mac16
-mmul16 -mno-mul16
-mmul32 -mno-mul32
-mnsa -mno-nsa
-mminmax -mno-minmax
-msext -mno-sext
-mbooleans -mno-booleans
-mhard-float -msoft-float
-mfused-madd -mno-fused-madd
-mserialize-volatile -mno-serialize-volatile
-mtext-section-literals -mno-text-section-literals
-mtarget-align -mno-target-align
-mlongcalls -mno-longcalls
```

Code Generation Options

See Section 3.18 [Options for Code Generation Conventions], page 140.

```
-fcall-saved-reg -fcall-used-reg
-ffixed-reg -fexceptions
-fnon-call-exceptions -funwind-tables
-fasynchronous-unwind-tables
-finhibit-size-directive -finstrument-functions
-fno-common -fno-ident -fno-gnu-linker
-fpcc-struct-return -fpic -fPIC
-freg-struct-return -fshared-data -fshort-enums
-fshort-double -fvolatile
-fvolatile-global -fvolatile-static
-fverbose-asm -fpack-struct -fstack-check
-fstack-limit-register=reg -fstack-limit-symbol=sym
-fargument-alias -fargument-noalias
-fargument-noalias-global -fleading-underscore
```

3.2 Options Controlling the Kind of Output

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

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

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

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

file`.ii` C++ source code which should not be preprocessed.

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

file.`mi` Objective-C source code which should not be preprocessed.

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

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

file.`f`
file.`for`
file.`FOR` Fortran source code which should not be preprocessed.

file.`F`
file.`fpp`
file.`FPP` Fortran source code which must be preprocessed (with the traditional preprocessor).

file.`r` Fortran source code which must be preprocessed with a RATFOR preprocessor (not included with GCC).

See section "Options Controlling the Kind of Output" in *Using and Porting GNU Fortran*, for more details of the handling of Fortran input files.

file.`ads` Ada source code file which contains a library unit declaration (a declaration of a package, subprogram, or generic, or a generic instantiation), or a library unit renaming declaration (a package, generic, or subprogram renaming declaration). Such files are also called *specs*.

file.`adb` Ada source code file containing a library unit body (a subprogram or package body). Such files are also called *bodies*.

file.`s` Assembler code.

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

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

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

`-x` *language*

Specify explicitly the *language* for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next '`-x`' option. Possible values for *language* are:

```
c  c-header  cpp-output
c++  c++-cpp-output
objective-c  objc-cpp-output
assembler  assembler-with-cpp
ada
f77  f77-cpp-input  ratfor
java
```

`-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).

`-pass-exit-codes`
: Normally the `gcc` program will exit with the code of 1 if any phase of the compiler returns a non-success return code. If you specify '`-pass-exit-codes`', the `gcc` program will instead return with numerically highest error produced by any phase that returned an error indication.

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

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

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

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

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

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

Input files that don't require compilation are ignored.

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

Input files which don't require preprocessing are ignored.

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

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

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

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

`-###` Like '`-v`' except the commands are not executed and all command arguments are quoted. This is useful for shell scripts to capture the driver-generated command lines.

`-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 is unable to read from a pipe; but the GNU assembler has no trouble.

`--help` Print (on the standard output) a description of the command line options understood by `gcc`. If the '`-v`' option is also specified then '`--help`' will also be passed on to the various processes invoked by `gcc`, so that they can display the command line options they accept. If the '`-W`' option is also specified then command line options which have no documentation associated with them will also be displayed.

`--target-help`
Print (on the standard output) a description of target specific command line options for each tool.

3.3 Compiling C++ Programs

C++ source files conventionally use one of the suffixes '`.C`', '`.cc`', '`.cpp`', '`.c++`', '`.cp`', or '`.cxx`'; preprocessed C++ files use the suffix '`.ii`'. GCC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name `gcc`).

However, C++ programs often require class libraries as well as a compiler that understands the C++ language—and under some circumstances, you might want to compile programs from standard input, or otherwise without a suffix that flags them as C++ programs. `g++` is a program that calls GCC with the default language set to C++, and automatically specifies linking against the C++ library. On many systems, `g++` is also installed with the name `c++`.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See Section 3.4 [Options Controlling C Dialect], page 19, for explanations of options for languages related to C. See Section 3.5 [Options Controlling C++ Dialect], page 24, for explanations of options that are meaningful only for C++ programs.

3.4 Options Controlling C Dialect

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

`-ansi` In C mode, support all ISO C89 programs. In C++ mode, remove GNU extensions that conflict with ISO C++.

This turns off certain features of GCC that are incompatible with ISO C89 (when compiling C code), or of standard C++ (when compiling C++ code), such as the `asm` 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 ISO trigraph feature. For the C compiler, it disables recognition of C++ style '`//`' comments as well as the `inline` keyword.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite '`-ansi`'. You would not want to use them in an ISO 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-ISO programs to be rejected gratuitously. For that, '`-pedantic`' is required in addition to '`-ansi`'. See Section 3.8 [Warning Options], page 31.

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 ISO standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

Functions which would normally be built in but do not have semantics defined by ISO C (such as `alloca` and `ffs`) are not built-in functions with '`-ansi`' is used. See Section 5.45 [Other built-in functions provided by GCC], page 217, for details of the functions affected.

`-std=` Determine the language standard. This option is currently only supported when compiling C. A value for this option must be provided; possible values are

'`c89`'
'`iso9899:1990`'
: ISO C89 (same as '`-ansi`').

'`iso9899:199409`'
: ISO C89 as modified in amendment 1.

'`c99`'
'`c9x`'
'`iso9899:1999`'
'`iso9899:199x`'
: ISO C99. Note that this standard is not yet fully supported; see `http://gcc.gnu.org/gcc-3.1/c99status.html` for more information. The names '`c9x`' and '`iso9899:199x`' are deprecated.

'`gnu89`'
: Default, ISO C89 plus GNU extensions (including some C99 features).

'`gnu99`'

'`gnu9x`'
: ISO C99 plus GNU extensions. When ISO C99 is fully implemented in GCC, this will become the default. The name '`gnu9x`' is deprecated.

Even when this option is not specified, you can still use some of the features of newer standards in so far as they do not conflict with previous C standards. For example, you may use `__restrict__` even when '`-std=c99`' is not specified.

The '`-std`' options specifying some version of ISO C have the same effects as '`-ansi`', except that features that were not in ISO C89 but are in the specified version (for example, '//' comments and the `inline` keyword in ISO C99) are not disabled.

See Chapter 2 [Language Standards Supported by GCC], page 5, for details of these standard versions.

`-aux-info` *filename*
: Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C.

  Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped ('`I`', '`N`' for new or '`O`' for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition ('`C`' or '`F`', respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.

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

  In C++, this switch only affects the `typeof` keyword, since `asm` and `inline` are standard keywords. You may want to use the '`-fno-gnu-keywords`' flag instead, which has the same effect. In C99 mode ('`-std=c99`' or '`-std=gnu99`'), this switch only affects the `asm` and `typeof` keywords, since `inline` is a standard keyword in ISO C99.

`-fno-builtin`
`-fno-builtin-`*function* (C and Objective-C only)
: Don't recognize built-in functions that do not begin with '`__builtin_`' as prefix. See Section 5.45 [Other built-in functions provided by GCC], page 217, for details of the functions affected, including those which are not built-in functions when '`-ansi`' or '`-std`' options for strict ISO C conformance are used because they do not have an ISO standard meaning.

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

  In C++, '`-fno-builtin`' is always in effect. The '`-fbuiltin`' option has no effect. Therefore, in C++, the only way to get the optimization benefits of built-in functions is to call the function using the '`__builtin_`' prefix. The GNU C++ Standard Library uses built-in functions to implement many functions (like `std::strchr`), so that you automatically get efficient code.

  With the '`-fno-builtin-`*function*' option, not available when compiling C++, only the built-in function *function* is disabled. *function* must not begin with '`__builtin_`'. If a function is named this is not built-in in this version of GCC, this option is ignored. There is no corresponding '-fbuiltin-*function*' option; if you wish to enable built-in functions selectively when using '-fno-builtin' or '-ffreestanding', you may define macros such as:

```
#define abs(n)          __builtin_abs ((n))
#define strcpy(d, s)    __builtin_strcpy ((d), (s))
```

-fhosted
: Assert that compilation takes place in a hosted environment. This implies '-fbuiltin'. A hosted environment is one in which the entire standard library is available, and in which `main` has a return type of `int`. Examples are nearly everything except a kernel. This is equivalent to '-fno-freestanding'.

-ffreestanding
: Assert that compilation takes place in a freestanding environment. This implies '-fno-builtin'. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at `main`. The most obvious example is an OS kernel. This is equivalent to '-fno-hosted'.

  See Chapter 2 [Language Standards Supported by GCC], page 5, for details of freestanding and hosted environments.

-trigraphs
: Support ISO C trigraphs. The '-ansi' option (and '-std' options for strict ISO C conformance) implies '-trigraphs'.

-no-integrated-cpp
: Invoke the external cpp during compilation. The default is to use the integrated cpp (internal cpp). This option also allows a user-supplied cpp via the '-B' option. This flag is applicable in both C and C++ modes.

  We do not guarantee to retain this option in future, and we may change its semantics.

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

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

- All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ISO C: automatic variables not declared `volatile` may be clobbered.
- The character escape sequences '`\x`' and '`\a`' evaluate as the literal characters '`x`' and '`a`' respectively. Without '`-traditional`', '`\x`' is a prefix for the hexadecimal representation of a character, and '`\a`' produces a bell.

This option is deprecated and may be removed.

You may wish to use '`-fno-builtin`' as well as '`-traditional`' if your program uses names that are normally GNU C built-in functions for other purposes of its own.

You cannot use '`-traditional`' if you include any header files that rely on ISO C features. Some vendors are starting to ship systems with ISO C header files and you cannot use '`-traditional`' on such systems to compile files that include any system headers.

The '`-traditional`' option also enables '`-traditional-cpp`'.

`-traditional-cpp`
: Attempt to support some aspects of traditional C preprocessors. See the GNU CPP manual for details.

`-fcond-mismatch`
: Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void. This option is not supported for C++.

`-funsigned-char`
: Let the type `char` be unsigned, like `unsigned char`.

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

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

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

`-fsigned-char`
: Let the type `char` be signed, like `signed char`.

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

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`

These options control whether a bit-field is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bit-field is signed, because this is consistent: the basic integer types such as `int` are signed types.

However, when '`-traditional`' is used, bit-fields are all unsigned no matter what.

`-fwritable-strings`

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

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

`-fallow-single-precision`

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

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

`-fshort-wchar`

Override the underlying type for '`wchar_t`' to be '`short unsigned int`' instead of the default for the target. This option is useful for building programs to run under WINE.

3.5 Options Controlling C++ Dialect

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

```
g++ -g -frepo -O -c firstClass.C
```

In this example, only '`-frepo`' is an option meant only for C++ programs; you can use the other options with any language supported by GCC.

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

`-fno-access-control`

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

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

  An alternative to using this option is to specify that your `operator new` does not throw any exceptions; if you declare it '`throw()`', G++ will check the return value. See also '`new (nothrow)`'.

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

  This option is no longer useful on most targets, now that support has been added for putting variables into BSS without making them common.

`-fno-const-strings`
: Give string constants type `char *` instead of type `const char *`. By default, G++ uses type `const char *` as required by the standard. Even if you use '`-fno-const-strings`', you cannot actually modify the value of a string constant, unless you also use '`-fwritable-strings`'.

  This option might be removed in a future release of G++. For maximum portability, you should structure your code so that it works with string constants that have type `const char *`.

`-fdollars-in-identifiers`
: Accept '`$`' in identifiers. You can also explicitly prohibit use of '`$`' with the option '`-fno-dollars-in-identifiers`'. (GNU C allows '`$`' by default on most target systems, but there are a few exceptions.) Traditional C allowed the character '`$`' to form part of identifiers. However, ISO C and C++ forbid '`$`' in identifiers.

`-fno-elide-constructors`
: The C++ standard allows an implementation to omit creating a temporary which is only used to initialize another object of the same type. Specifying this option disables that optimization, and forces G++ to call the copy constructor in all cases.

`-fno-enforce-eh-specs`
: Don't check for violation of exception specifications at runtime. This option violates the C++ standard, but may be useful for reducing code size in production builds, much like defining '`NDEBUG`'. The compiler will still optimize based on the exception specifications.

`-fexternal-templates`
: Cause '`#pragma interface`' and '`implementation`' to apply to template instantiation; template instances are emitted or not according to the location of the template definition. See Section 6.6 [Template Instantiation], page 254, for more information.

This option is deprecated.

`-falt-external-templates`
: Similar to '`-fexternal-templates`', but template instances are emitted or not according to the place where they are first instantiated. See Section 6.6 [Template Instantiation], page 254, for more information.

This option is deprecated.

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

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

`-fno-gnu-keywords`
: Do not recognize `typeof` as a keyword, so that code can use this word as an identifier. You can use the keyword `__typeof__` instead. '`-ansi`' implies '`-fno-gnu-keywords`'.

`-fno-implicit-templates`
: Never emit code for non-inline templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Section 6.6 [Template Instantiation], page 254, for more information.

`-fno-implicit-inline-templates`
: Don't emit code for implicit instantiations of inline templates, either. The default is to handle inlines differently so that compiles with and without optimization will need the same set of explicit instantiations.

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

`-fms-extensions`
: Disable pedantic warnings about constructs used in MFC, such as implicit int and getting a pointer to member function via non-standard syntax.

`-fno-nonansi-builtins`
: Disable built-in declarations of functions that are not mandated by ANSI/ISO C. These include `ffs`, `alloca`, `_exit`, `index`, `bzero`, `conjf`, and other related functions.

`-fno-operator-names`
: Do not treat the operator name keywords `and`, `bitand`, `bitor`, `compl`, `not`, `or` and `xor` as synonyms as keywords.

`-fno-optional-diags`
: Disable diagnostics that the standard says a compiler does not need to issue. Currently, the only such diagnostic issued by G++ is the one for a name having multiple meanings within a class.

`-fpermissive`
: Downgrade messages about nonconformant code from errors to warnings. By default, G++ effectively sets '`-pedantic-errors`' without '`-pedantic`'; this option reverses that. This behavior and this option are superseded by '`-pedantic`', which works as it does for GNU C.

`-frepo`
: Enable automatic template instantiation at link time. This option also implies '`-fno-implicit-templates`'. See Section 6.6 [Template Instantiation], page 254, for more information.

`-fno-rtti`
: Disable generation of information about every class with virtual functions for use by the C++ runtime type identification features ('`dynamic_cast`' and '`typeid`'). If you don't use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed.

`-fstats`
: Emit statistics about front-end processing at the end of the compilation. This information is generally only useful to the G++ development team.

`-ftemplate-depth-`*n*
: Set the maximum instantiation depth for template classes to *n*. A limit on the template instantiation depth is needed to detect endless recursions during template class instantiation. ANSI/ISO C++ conforming programs must not rely on a maximum depth greater than 17.

`-fuse-cxa-atexit`
: Register destructors for objects with static storage duration with the `__cxa_atexit` function rather than the `atexit` function. This option is required for fully standards-compliant handling of static destructors, but will only work if your C library supports `__cxa_atexit`.

`-fvtable-gc`
: Emit special relocations for vtables and virtual function references so that the linker can identify unused virtual functions and zero out vtable slots that refer to them. This is most useful with '`-ffunction-sections`' and '`-Wl,--gc-sections`', in order to also discard the functions themselves.

  This optimization requires GNU as and GNU ld. Not all systems support this option. '`-Wl,--gc-sections`' is ignored without '`-static`'.

`-fno-weak`
: Do not use weak symbol support, even if it is provided by the linker. By default, G++ will use weak symbols if they are available. This option exists only for testing, and should not be used by end-users; it will result in inferior code and has no benefits. This option may be removed in a future release of G++.

`-nostdinc++`

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

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

`-fno-default-inline`

Do not assume '`inline`' for functions defined inside a class scope. See Section 3.10 [Options That Control Optimization], page 50. Note that these functions will have linkage like inline functions; they just won't be inlined by default.

`-Wctor-dtor-privacy` (C++ only)

Warn when a class seems unusable, because all the constructors or destructors in a class are private and the class has no friends or public static member functions.

`-Wnon-virtual-dtor` (C++ only)

Warn when a class declares a non-virtual destructor that should probably be virtual, because it looks like the class will be used polymorphically.

`-Wreorder` (C++ only)

Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

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

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

The following '`-W...`' options are not affected by '`-Wall`'.

`-Weffc++` (C++ only)

Warn about violations of the following style guidelines from Scott Meyers' *Effective C++* book:

- Item 11: Define a copy constructor and an assignment operator for classes with dynamically allocated memory.
- Item 12: Prefer initialization to assignment in constructors.
- Item 14: Make destructors virtual in base classes.
- Item 15: Have `operator=` return a reference to `*this`.
- Item 23: Don't try to return a reference when you must return an object.

and about violations of the following style guidelines from Scott Meyers' *More Effective C++* book:

- Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.
- Item 7: Never overload `&&`, `||`, or `,`.

If you use this option, you should be aware that the standard library headers do not obey all of these guidelines; you can use '`grep -v`' to filter out those warnings.

`-Wno-deprecated` (C++ only)
: Do not warn about usage of deprecated features. See Section 6.10 [Deprecated Features], page 258.

`-Wno-non-template-friend` (C++ only)
: Disable warnings when non-templatized friend functions are declared within a template. With the advent of explicit template specification support in G++, if the name of the friend is an unqualified-id (i.e., '`friend foo(int)`'), the C++ language specification demands that the friend declare or define an ordinary, nontemplate function. (Section 14.5.3). Before G++ implemented explicit specification, unqualified-ids could be interpreted as a particular specialization of a templatized function. Because this non-conforming behavior is no longer the default behavior for G++, '`-Wnon-template-friend`' allows the compiler to check existing code for potential trouble spots, and is on by default. This new compiler behavior can be turned off with '`-Wno-non-template-friend`' which keeps the conformant compiler code but disables the helpful warning.

`-Wold-style-cast` (C++ only)
: Warn if an old-style (C-style) cast to a non-void type is used within a C++ program. The new-style casts ('`static_cast`', '`reinterpret_cast`', and '`const_cast`') are less vulnerable to unintended effects, and much easier to grep for.

`-Woverloaded-virtual` (C++ only)
: Warn when a function declaration hides virtual functions from a base class. For example, in:

```
struct A {
  virtual void f();
};

struct B: public A {
  void f(int);
};
```

the `A` class version of `f` is hidden in `B`, and code like this:

```
B* b;
b->f();
```

will fail to compile.

`-Wno-pmf-conversions` (C++ only)
: Disable the diagnostic for converting a bound pointer to member function to a plain pointer.

`-Wsign-promo` (C++ only)
: Warn when overload resolution chooses a promotion from unsigned or enumeral type to a signed type over a conversion to an unsigned type of the same size.

Previous versions of G++ would try to preserve unsignedness, but the standard mandates the current behavior.

`-Wsynth` (C++ only)
: Warn when G++'s synthesis behavior does not match that of cfront. For instance:

```
struct A {
  operator int ();
  A& operator = (int);
};

main ()
{
  A a,b;
  a = b;
}
```

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

3.6 Options Controlling Objective-C Dialect

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

```
gcc -g -fgnu-runtime -O -c some_class.m
```

In this example, only '`-fgnu-runtime`' is an option meant only for Objective-C programs; you can use the other options with any language supported by GCC.

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

`-fconstant-string-class=`*class-name*
: Use *class-name* as the name of the class to instantiate for each literal string specified with the syntax `@"..."`. The default class name is `NXConstantString`.

`-fgnu-runtime`
: Generate object code compatible with the standard GNU Objective-C runtime. This is the default for most types of systems.

`-fnext-runtime`
: Generate output compatible with the NeXT runtime. This is the default for NeXT-based systems, including Darwin and Mac OS X.

`-gen-decls`
: Dump interface declarations for all classes seen in the source file to a file named '*sourcename*`.decl`'.

`-Wno-protocol`
: Do not warn if methods required by a protocol are not implemented in the class adopting it.

`-Wselector`
: Warn if a selector has multiple methods of different types defined.

3.7 Options to Control Diagnostic Messages Formatting

Traditionally, diagnostic messages have been formatted irrespective of the output device's aspect (e.g. its width, ...). The options described below can be used to control the diagnostic messages formatting algorithm, e.g. how many characters per line, how often source location information should be reported. Right now, only the C++ front end can honor these options. However it is expected, in the near future, that the remaining front ends would be able to digest them correctly.

`-fmessage-length=`*n*
: Try to format error messages so that they fit on lines of about *n* characters. The default is 72 characters for `g++` and 0 for the rest of the front ends supported by GCC. If *n* is zero, then no line-wrapping will be done; each error message will appear on a single line.

`-fdiagnostics-show-location=once`
: Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit *once* source location information; that is, in case the message is too long to fit on a single physical line and has to be wrapped, the source location won't be emitted (as prefix) again, over and over, in subsequent continuation lines. This is the default behavior.

`-fdiagnostics-show-location=every-line`
: Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit the same source location information (as prefix) for physical lines that result from the process of breaking a message which is too long to fit on a single line.

3.8 Options to Request or Suppress Warnings

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

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

The following options control the amount and kinds of warnings produced by GCC; for further, language-specific options also refer to Section 3.5 [C++ Dialect Options], page 24 and Section 3.6 [Objective-C Dialect Options], page 30.

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

`-pedantic`
: Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any '`-std`' option used.

Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few will require '`-ansi`' or a '`-std`' option specifying the required version of ISO C). However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are rejected.

'`-pedantic`' does not cause warning messages for use of the alternate keywords whose names begin and end with '`__`'. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them. See Section 5.40 [Alternate Keywords], page 213.

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

A feature to report any failure to conform to ISO C might be useful in some instances, but would require considerable additional work and would be quite different from '`-pedantic`'. We don't have plans to support such a feature in the near future.

Where the standard specified with '`-std`' represents a GNU extended dialect of C, such as '`gnu89`' or '`gnu99`', there is a corresponding *base standard*, the version of ISO C on which the GNU extended dialect is based. Warnings from '`-pedantic`' are given where they are required by the base standard. (It would not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

`-pedantic-errors`
: Like '`-pedantic`', except that errors are produced rather than warnings.

`-w`
: Inhibit all warning messages.

`-Wno-import`
: Inhibit warning messages about the use of '`#import`'.

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

`-Wcomment`
: Warn whenever a comment-start sequence '`/*`' appears in a '`/*`' comment, or whenever a Backslash-Newline appears in a '`//`' comment.

`-Wformat`
: Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense. This includes standard functions, and others specified by format attributes (see Section 5.26 [Function Attributes], page 172), in the `printf`, `scanf`, `strftime` and `strfmon` (an X/Open extension, not in the C standard) families.

The formats are checked against the format features supported by GNU libc version 2.2. These include all ISO C89 and C99 features, as well as features from the Single Unix Specification and some BSD and GNU extensions. Other library implementations may not support all these features; GCC does not support warning about features that go beyond a particular library's limitations. However, if '-pedantic' is used with '-Wformat', warnings will be given about format features not in the selected standard version (but not for strfmon formats, since those are not in any version of the C standard). See Section 3.4 [Options Controlling C Dialect], page 19.

'-Wformat' is included in '-Wall'. For more control over some aspects of format checking, the options '-Wno-format-y2k', '-Wno-format-extra-args', '-Wformat-nonliteral', '-Wformat-security' and '-Wformat=2' are available, but are not included in '-Wall'.

-Wno-format-y2k

If '-Wformat' is specified, do not warn about strftime formats which may yield only a two-digit year.

-Wno-format-extra-args

If '-Wformat' is specified, do not warn about excess arguments to a printf or scanf format function. The C standard specifies that such arguments are ignored.

Where the unused arguments lie between used arguments that are specified with '$' operand number specifications, normally warnings are still given, since the implementation could not know what type to pass to va_arg to skip the unused arguments. However, in the case of scanf formats, this option will suppress the warning if the unused arguments are all pointers, since the Single Unix Specification says that such unused arguments are allowed.

-Wformat-nonliteral

If '-Wformat' is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a va_list.

-Wformat-security

If '-Wformat' is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to printf and scanf functions where the format string is not a string literal and there are no format arguments, as in printf (foo);. This may be a security hole if the format string came from untrusted input and contains '%n'. (This is currently a subset of what '-Wformat-nonliteral' warns about, but in future warnings may be added to '-Wformat-security' that are not included in '-Wformat-nonliteral'.)

-Wformat=2

Enable '-Wformat' plus format checks not included in '-Wformat'. Currently equivalent to '-Wformat -Wformat-nonliteral -Wformat-security'.

-Wimplicit-int

Warn when a declaration does not specify a type.

`-Wimplicit-function-declaration`
`-Werror-implicit-function-declaration`
Give a warning (or error) whenever a function is used before being declared.

`-Wimplicit`
Same as '`-Wimplicit-int`' and '`-Wimplicit-function-declaration`'.

`-Wmain`
Warn if the type of '`main`' is suspicious. '`main`' should be a function with external linkage, returning int, taking either zero arguments, two, or three arguments of appropriate types.

`-Wmissing-braces`
Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for '`a`' is not fully bracketed, but that for '`b`' is fully bracketed.

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 0, 1 }, { 2, 3 } };
```

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

Also warn about constructions where there may be confusion to which `if` statement an `else` branch belongs. Here is an example of such a case:

```
{
  if (a)
    if (b)
      foo ();
  else
    bar ();
}
```

In C, every `else` branch belongs to the innermost possible `if` statement, which in this example is `if (b)`. This is often not what the programmer expected, as illustrated in the above example by indentation the programmer chose. When there is the potential for this confusion, GCC will issue a warning when this flag is specified. To eliminate the warning, add explicit braces around the innermost `if` statement so there is no way the `else` could belong to the enclosing `if`. The resulting code would look like this:

```
{
  if (a)
    {
      if (b)
        foo ();
      else
        bar ();
    }
}
```

`-Wsequence-point`
: Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard.

The C standard defines the order in which expressions in a C program are evaluated in terms of *sequence points*, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.". If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

The present implementation of this option only works for C programs. A future implementation may also work for C++ programs.

The C standard is worded confusingly, therefore there is some debate over the precise meaning of the sequence point rules in subtle cases. Links to discussions of the problem, including proposed formal definitions, may be found on our readings page, at `http://gcc.gnu.org/readings.html`.

`-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`.

For C++, a function without return type always produces a diagnostic message, even when '`-Wno-return-type`' is specified. The only exceptions are '`main`' and functions defined in system headers.

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

`-Wtrigraphs`
: Warn if any trigraphs are encountered that might change the meaning of the program (trigraphs within comments are not warned about).

`-Wunused-function`
: Warn whenever a static function is declared but not defined or a non\-inline static function is unused.

`-Wunused-label`
: Warn whenever a label is declared but not used.

  To suppress this warning use the '`unused`' attribute (see Section 5.33 [Variable Attributes], page 184).

`-Wunused-parameter`
: Warn whenever a function parameter is unused aside from its declaration.

  To suppress this warning use the '`unused`' attribute (see Section 5.33 [Variable Attributes], page 184).

`-Wunused-variable`
: Warn whenever a local variable or non-constant static variable is unused aside from its declaration

  To suppress this warning use the '`unused`' attribute (see Section 5.33 [Variable Attributes], page 184).

`-Wunused-value`
: Warn whenever a statement computes a result that is explicitly not used.

  To suppress this warning cast the expression to '`void`'.

`-Wunused`
: All all the above '`-Wunused`' options combined.

  In order to get a warning about an unused function parameter, you must either specify '`-W -Wunused`' or separately specify '`-Wunused-parameter`'.

`-Wuninitialized`
: Warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a `setjmp` call.

  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 GCC 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 GCC 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.

This option also warns when a non-volatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

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

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 5.26 [Function Attributes], page 172.

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

`-Wunknown-pragmas`
: Warn when a #pragma directive is encountered which is not understood by GCC. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the '`-Wall`' command line option.

`-Wall`
: All of the above '`-W`' options combined. This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

`-Wdiv-by-zero`
: Warn about compile-time integer division by zero. This is default. To inhibit the warning messages, use '`-Wno-div-by-zero`'. Floating point division by zero

is not warned about, as it can be a legitimate way of obtaining infinities and NaNs.

`-Wmultichar`
: Warn if a multicharacter constant (‘'FOOF'’) is used. This is default. To inhibit the warning messages, use ‘-Wno-multichar’. Usually they indicate a typo in the user's code, as they have implementation-defined values, and should not be used in portable code.

`-Wsystem-headers`
: Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells GCC to emit warnings from system headers as if they occurred in user code. However, note that using ‘-Wall’ in conjunction with this option will *not* warn about unknown pragmas in system headers—for that, ‘-Wunknown-pragmas’ must also be used.

The following ‘-W...’ options are not implied by ‘-Wall’. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

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

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

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

- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as ‘x[i,j]’ will cause a warning, but ‘x[(void)i,j]’ will not.
- An unsigned value is compared against zero with ‘<’ or ‘<=’.
- A comparison like ‘x<=y<=z’ appears; this is equivalent to ‘(x<=y ? 1 : 0) <= z’, which is a different interpretation from that of ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- The return type of a function has a type qualifier such as `const`. Such a type qualifier has no effect, since the value returned by a function is not an lvalue. (But don't warn about the GNU extension of `volatile void` return types. That extension will be warned about if ‘-pedantic’ is specified.)
- If ‘-Wall’ or ‘-Wunused’ is also specified, warn about unused arguments.

- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (But don't warn if '-Wno-sign-compare' is also specified.)
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

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

- An aggregate has an initializer which does not initialize all members. For example, the following code would cause such a warning, because `x.h` would be implicitly initialized to zero:

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```

`-Wfloat-equal`

Warn if floating point values are used in equality comparisons.

The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analysing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you would check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

`-Wtraditional` (C only)

Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and/or problematic constructs which should be avoided.

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but does not in ISO C.
- In traditional C, some preprocessor directives did not exist. Traditional preprocessors would only consider a line to be a directive if the '#' appeared in column 1 on the line. Therefore '-Wtraditional' warns about directives that traditional C understands but would ignore because the '#' does not appear as the first character on the line. It also suggests you hide directives like '#pragma' not understood by traditional C by indenting them. Some traditional implementations would not recognize '#elif', so it suggests avoiding it altogether.
- A function-like macro that appears without arguments.
- The unary plus operator.
- The 'U' integer constant suffix, or the 'F' or 'L' floating point constant suffixes. (Traditional C does support the 'L' suffix on integer constants.)

Note, these suffixes appear in macros defined in the system headers of most modern systems, e.g. the '_MIN'/'_MAX' macros in <limits.h>. Use of these macros in user code might normally lead to spurious warnings, however gcc's integrated preprocessor has enough context to avoid warning in these cases.

- 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`.
- A non-`static` function declaration follows a `static` one. This construct is not accepted by some traditional C compilers.
- The ISO type of an integer constant has a different width or signedness from its traditional type. This warning is only issued if the base of the constant is ten. I.e. hexadecimal or octal values, which typically represent bit patterns, are not warned about.
- Usage of ISO string concatenation is detected.
- Initialization of automatic aggregates.
- Identifier conflicts with labels. Traditional C lacks a separate namespace for labels.
- Initialization of unions. If the initializer is zero, the warning is omitted. This is done under the assumption that the zero initializer in user code appears conditioned on e.g. `__STDC__` to avoid missing initializer warnings and relies on default initialization to zero in the traditional C case.
- Conversions by prototypes between fixed/floating point values and vice versa. The absence of these prototypes when compiling with traditional C would cause serious problems. This is a subset of the possible conversion warnings, for the full set use '-Wconversion'.

`-Wundef` Warn if an undefined identifier is evaluated in an '#if' directive.

`-Wshadow` Warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed.

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

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

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

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

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

`-Wwrite-strings`
When compiling C, 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; when compiling C++, warn about the deprecated conversion from string constants to `char *`. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make '`-Wall`' request these warnings.

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

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

`-Wsign-compare`
Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by '`-W`'; to get the other warnings of '`-W`' without this warning, use '`-W -Wno-sign-compare`'.

`-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` (C only)
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` (C only)
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.

`-Wmissing-noreturn`
Warn about functions which might be candidates for attribute `noreturn`. Note these are only possible candidates, not absolute ones. Care should be taken

to manually verify functions actually do not ever return before adding the `noreturn` attribute, otherwise subtle code generation bugs could be introduced. You will not get a warning for `main` in hosted C environments.

`-Wmissing-format-attribute`
: If '`-Wformat`' is enabled, also warn about functions which might be candidates for `format` attributes. Note these are only possible candidates, not absolute ones. GCC will guess that `format` attributes might be appropriate for any function that calls a function like `vprintf` or `vscanf`, but this might not always be the case, and some functions for which `format` attributes are appropriate may not be detected. This option has no effect unless '`-Wformat`' is enabled (possibly by '`-Wall`').

`-Wno-deprecated-declarations`
: Do not warn about uses of functions, variables, and types marked as deprecated by using the `deprecated` attribute. (see Section 5.26 [Function Attributes], page 172, see Section 5.33 [Variable Attributes], page 184, see Section 5.34 [Type Attributes], page 188.)

`-Wpacked`
: Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure. Such structures may be misaligned for little benefit. For instance, in this code, the variable `f.x` in `struct bar` will be misaligned even though `struct bar` does not itself have the packed attribute:

```
struct foo {
  int x;
  char a, b, c, d;
} __attribute__((packed));
struct bar {
  char z;
  struct foo f;
};
```

`-Wpadded`
: Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller.

`-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` (C only)
: Warn if an `extern` declaration is encountered within a function.

`-Wunreachable-code`
: Warn if the compiler detects that code will never be executed.

: This option is intended to warn when the compiler detects that at least a whole line of source code will never be executed, because some condition is never satisfied or because it is after a procedure that never returns.

It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently-unreachable code.

For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.

This option is not made part of '-Wall' because in a debugging version of a program there is often substantial code which checks correct functioning of the program and is, hopefully, unreachable because the program does work. Another common use of unreachable code is to provide behavior which is selectable at compile-time.

`-Winline` Warn if a function can not be inlined and it was declared as inline.

`-Wlong-long`
: Warn if 'long long' type is used. This is default. To inhibit the warning messages, use '-Wno-long-long'. Flags '-Wlong-long' and '-Wno-long-long' are taken into account only when '-pedantic' flag is used.

`-Wdisabled-optimization`
: Warn if a requested optimization pass is disabled. This warning does not generally indicate that there is anything wrong with your code; it merely indicates that GCC's optimizers were unable to handle the code effectively. Often, the problem is that your code is too big or too complex; GCC will refuse to optimize programs when the optimization itself is likely to take inordinate amounts of time.

`-Werror` Make all warnings into errors.

3.9 Options for Debugging Your Program or GCC

GCC 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.

On most systems that use stabs format, '-g' enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use '-gstabs+', '-gstabs', '-gxcoff+', '-gxcoff', '-gdwarf-1+', '-gdwarf-1', or '-gvms' (see below).

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

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

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

`-ggdb` Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF 2, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

`-gstabs` Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output which is not understood by DBX or SDB. On System V Release 4 systems this option requires the GNU assembler.

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

`-gcoff` Produce debugging information in COFF format (if that is supported). This is the format used by SDB on most System V systems prior to System V Release 4.

`-gxcoff` Produce debugging information in XCOFF format (if that is supported). This is the format used by the DBX debugger on IBM RS/6000 systems.

`-gxcoff+` Produce debugging information in XCOFF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program, and may cause assemblers other than the GNU assembler (GAS) to fail with an error.

`-gdwarf` Produce debugging information in DWARF version 1 format (if that is supported). This is the format used by SDB on most System V Release 4 systems.

`-gdwarf+` Produce debugging information in DWARF version 1 format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-gdwarf-2`
Produce debugging information in DWARF version 2 format (if that is supported). This is the format used by DBX on IRIX 6.

`-gvms` Produce debugging information in VMS debug format (if that is supported). This is the format used by DEBUG on VMS systems.

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gcoff`*level*
`-gxcoff`*level*
`-gvms`*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`'.

Note that in order to avoid confusion between DWARF1 debug level 2, and DWARF2, neither '`-gdwarf`' nor '`-gdwarf-2`' accept a concatenated debug level. Instead use an additional '`-g`*level*' option to change the debug level for DWARF1 or DWARF2.

`-p` Generate extra code to write profile information suitable for the analysis program `prof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

`-pg` Generate extra code to write profile information suitable for the analysis program `gprof`. You must use this option when compiling the source files you want data about, and you must also use it when linking.

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

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

`-Q` Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.

`-ftime-report`

Makes the compiler print some statistics about the time consumed by each pass when it finishes.

`-fmem-report`

Makes the compiler print some statistics about permanent memory allocation when it finishes.

`-fprofile-arcs`

Instrument *arcs* during compilation to generate coverage data or for profile-directed block ordering. During execution the program records how many times each branch is executed and how many times it is taken. When the compiled program exits it saves this data to a file called '*sourcename*`.da`' for each source file.

For profile-directed block ordering, compile the program with '`-fprofile-arcs`' plus optimization and code generation options, generate the arc profile information by running the program on a selected workload, and then compile the program again with the same optimization and code generation options plus

'-fbranch-probabilities' (see Section 3.10 [Options that Control Optimization], page 50).

The other use of '-fprofile-arcs' is for use with gcov, when it is used with the '-ftest-coverage' option. GCC supports two methods of determining code coverage: the options that support gcov, and options '-a' and '-ax', which write information to text files. The options that support gcov do not need to instrument every arc in the program, so a program compiled with them runs faster than a program compiled with '-a', which adds instrumentation code to every basic block in the program. The tradeoff: since gcov does not have execution counts for all branches, it must start with the execution counts for the instrumented branches, and then iterate over the program flow graph until the entire graph has been solved. Hence, gcov runs a little more slowly than a program which uses information from '-a' and '-ax'.

With '-fprofile-arcs', for each function of your program GCC creates a program flow graph, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

This option makes it possible to estimate branch probabilities and to calculate basic block execution counts. In general, basic block execution counts as provided by '-a' do not give enough information to estimate all branch probabilities.

`-ftest-coverage`

Create data files for the gcov code-coverage utility (see Chapter 8 [gcov: a GCC Test Coverage Program], page 267). The data file names begin with the name of your source file:

sourcename.bb

A mapping from basic blocks to line numbers, which gcov uses to associate basic block execution counts with line numbers.

sourcename.bbg

A list of all arcs in the program flow graph. This allows gcov to reconstruct the program flow graph, so that it can compute all basic block and arc execution counts from the information in the *sourcename*.da file.

Use '-ftest-coverage' with '-fprofile-arcs'; the latter option adds instrumentation to the program, which then writes execution counts to another data file:

sourcename.da

Runtime arc execution counts, used in conjunction with the arc information in the file *sourcename*.bbg.

Coverage data will map better to the source files if '-ftest-coverage' is used without optimization.

-d*letters* Says to make debugging dumps during compilation at times specified by *letters*. This is used for debugging the compiler. The file names for most of the dumps are made by appending a pass number and a word to the source file name (e.g. 'foo.c.00.rtl' or 'foo.c.01.sibling'). Here are the possible letters for use in *letters*, and their meanings:

'A' Annotate the assembler output with miscellaneous debugging information.

'b' Dump after computing branch probabilities, to '*file*.14.bp'.

'B' Dump after block reordering, to '*file*.29.bbro'.

'c' Dump after instruction combination, to the file '*file*.16.combine'.

'C' Dump after the first if conversion, to the file '*file*.17.ce'.

'd' Dump after delayed branch scheduling, to '*file*.31.dbr'.

'D' Dump all macro definitions, at the end of preprocessing, in addition to normal output.

'e' Dump after SSA optimizations, to '*file*.04.ssa' and '*file*.07.ussa'.

'E' Dump after the second if conversion, to '*file*.26.ce2'.

'f' Dump after life analysis, to '*file*.15.life'.

'F' Dump after purging ADDRESSOF codes, to '*file*.09.addressof'.

'g' Dump after global register allocation, to '*file*.21.greg'.

'h' Dump after finalization of EH handling code, to '*file*.02.eh'.

'k' Dump after reg-to-stack conversion, to '*file*.28.stack'.

'o' Dump after post-reload optimizations, to '*file*.22.postreload'.

'G' Dump after GCSE, to '*file*.10.gcse'.

'i' Dump after sibling call optimizations, to '*file*.01.sibling'.

'j' Dump after the first jump optimization, to '*file*.03.jump'.

'k' Dump after conversion from registers to stack, to '*file*.32.stack'.

'l' Dump after local register allocation, to '*file*.20.lreg'.

'L' Dump after loop optimization, to '*file*.11.loop'.

'M' Dump after performing the machine dependent reorganisation pass, to '*file*.30.mach'.

'n' Dump after register renumbering, to '*file*.25.rnreg'.

'N' Dump after the register move pass, to '*file*.18.regmove'.

'r' Dump after RTL generation, to '*file*.00.rtl'.

'R' Dump after the second scheduling pass, to '*file*.27.sched2'.

's' Dump after CSE (including the jump optimization that sometimes follows CSE), to '*file*.08.cse'.

'S' Dump after the first scheduling pass, to '*file*.19.sched'.

't' Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to '*file*.12.cse2'.

'w' Dump after the second flow pass, to '*file*.23.flow2'.

'X' Dump after SSA dead code elimination, to '*file*.06.ssadce'.

'z' Dump after the peephole pass, to '*file*.24.peephole2'.

'a' Produce all the dumps listed above.

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

'p' Annotate the assembler output with a comment indicating which pattern and alternative was used. The length of each instruction is also printed.

'P' Dump the RTL in the assembler output as a comment before each instruction. Also turns on '-dp' annotation.

'v' For each of the other indicated dump files (except for '*file*.00.rtl'), dump a representation of the control flow graph suitable for viewing with VCG to '*file*.*pass*.vcg'.

'x' Just generate RTL for a function instead of compiling it. Usually used with 'r'.

'y' Dump debugging information during parsing, to standard error.

`-fdump-unnumbered`
: When doing debugging dumps (see '-d' option above), suppress instruction numbers and line number note output. This makes it more feasible to use diff on debugging dumps for compiler invocations with different options, in particular with and without '-g'.

`-fdump-translation-unit` (C and C++ only)
`-fdump-translation-unit-`*options* (C and C++ only)
: Dump a representation of the tree structure for the entire translation unit to a file. The file name is made by appending '.tu' to the source file name. If the '-*options*' form is used, *options* controls the details of the dump as described for the '-fdump-tree' options.

`-fdump-class-hierarchy` (C++ only)
`-fdump-class-hierarchy-`*options* (C++ only)
: Dump a representation of each class's hierarchy and virtual function table layout to a file. The file name is made by appending '.class' to the source file name. If the '-*options*' form is used, *options* controls the details of the dump as described for the '-fdump-tree' options.

`-fdump-tree-`*switch* (C++ only)
`-fdump-tree-`*switch*-*options* (C++ only)
: Control the dumping at various stages of processing the intermediate language tree to a file. The file name is generated by appending a switch specific suffix

to the source file name. If the '-*options*' form is used, *options* is a list of '-' separated options that control the details of the dump. Not all options are applicable to all dumps, those which are not meaningful will be ignored. The following options are available

'address' Print the address of each node. Usually this is not meaningful as it changes according to the environment and source file. Its primary use is for tying up a dump file with a debug environment.

'slim' Inhibit dumping of members of a scope or body of a function merely because that scope has been reached. Only dump such items when they are directly reachable by some other path.

'all' Turn on all options.

The following tree dumps are possible:

'original'
: Dump before any tree based optimization, to '*file*.original'.

'optimized'
: Dump after all tree based optimization, to '*file*.optimized'.

'inlined' Dump after function inlining, to '*file*.inlined'.

-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 GCC would make when running on the target machine.

-save-temps
: Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling 'foo.c' with '-c -save-temps' would produce files 'foo.i' and 'foo.s', as well as 'foo.o'. This creates a preprocessed 'foo.i' output file even though the compiler now normally uses an integrated preprocessor.

-time Report the CPU time taken by each subprocess in the compilation sequence. For C source files, this is the compiler proper and assembler (plus the linker if linking is done). The output looks like this:

```
# cc1 0.12 0.01
# as 0.00 0.01
```

The first number on each line is the "user time," that is time spent executing the program itself. The second number is "system time," time spent executing operating system routines on behalf of the program. Both numbers are in seconds.

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

`-print-multi-directory`
: Print the directory name corresponding to the multilib selected by any other switches present in the command line. This directory is supposed to exist in `GCC_EXEC_PREFIX`.

`-print-multi-lib`
: Print the mapping from multilib directory names to compiler switches that enable them. The directory name is separated from the switches by '`;`', and each switch starts with an '`@`' instead of the '`-`', without spaces between multiple switches. This is supposed to ease shell-processing.

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

`-print-libgcc-file-name`
: Same as '`-print-file-name=libgcc.a`'.

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

```
gcc -nostdlib files... `gcc -print-libgcc-file-name`
```

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

This is useful when gcc prints the error message '`installation problem, cannot exec cpp0: No such file or directory`'. To resolve this you either need to put '`cpp0`' and the other compiler components where gcc expects to find them, or you can set the environment variable `GCC_EXEC_PREFIX` to the directory where you installed them. Don't forget the trailing '/'. See Section 3.19 [Environment Variables], page 145.

`-dumpmachine`
: Print the compiler's target machine (for example, '`i686-pc-linux-gnu`')—and don't do anything else.

`-dumpversion`
: Print the compiler version (for example, '`3.0`')—and don't do anything else.

`-dumpspecs`
: Print the compiler's built-in specs—and don't do anything else. (This is used when GCC itself is being built.) See Section 3.15 [Spec Files], page 73.

3.10 Options That Control Optimization

These options control various sorts of optimizations:

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

Without '`-O`', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent:

if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

With '-O', the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

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

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

Please note the warning under '-fgcse' about invoking '-O2' on programs that use computed gotos.

`-O3` Optimize yet more. '-O3' turns on all optimizations specified by '-O2' and also turns on the '-finline-functions' and '-frename-registers' options.

`-O0` Do not optimize.

`-Os` Optimize for size. '-Os' enables all '-O2' optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

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

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

`-ffloat-store`

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

This option 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. Similarly for the x86 architecture. 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, after modifying them to store all pertinent intermediate computations into variables.

`-fno-default-inline`

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

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

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

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

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

  On some machines, such as the VAX, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro `FRAME_POINTER_REQUIRED` controls whether a target machine supports this flag. See section "Register Usage" in *GNU Compiler Collection (GCC) Internals*.

`-foptimize-sibling-calls`
: Optimize sibling and tail recursive calls.

`-ftrapv`
: This option generates traps for signed overflow on addition, subtraction, multiplication operations.

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

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

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

`-finline-limit=`*n*
: By default, gcc limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (ie marked with the inline keyword or defined within the class definition in c++). *n* is the size of functions that can be inlined in number of pseudo instructions

(not counting parameter handling). The default value of *n* is 600. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining heavily such as those based on recursive templates with C++.

Note: pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way, it represents a count of assembly instructions and as such its exact meaning might change from one release to an another.

`-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. This switch does not affect `extern inline` functions.

`-fkeep-static-consts`

Emit variables declared `static const` when optimization isn't turned on, even if the variables aren't referenced.

GCC enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the '`-fno-keep-static-consts`' option.

`-fmerge-constants`

Attempt to merge identical constants (string constants and floating point constants) accross compilation units.

This option is default for optimized compilation if assembler and linker support it. Use '`-fno-merge-constants`' to inhibit this behavior.

`-fmerge-all-constants`

Attempt to merge identical constants and identical variables.

This option implies '`-fmerge-constants`'. In addition to '`-fmerge-constants`' this considers e.g. even constant initialized arrays or initialized constant variables with integral or floating point types. Languages like C or C++ require each non-automatic variable to have distinct location, so using this option will result in non-conforming behavior.

`-fno-function-cse`

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

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

`-ffast-math`

Sets '`-fno-math-errno`', '`-funsafe-math-optimizations`', and '`-fno-trapping-math`'.

This option causes the preprocessor macro `__FAST_MATH__` to be defined.

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

`-fno-math-errno`
: Do not set ERRNO after calling math functions that are executed with a single instruction, e.g., sqrt. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.

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

  The default is '`-fmath-errno`'.

`-funsafe-math-optimizations`
: Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. When used at link-time, it may include libraries or startup files that change the default FPU control word or other similar optimizations.

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

  The default is '`-fno-unsafe-math-optimizations`'.

`-fno-trapping-math`
: Compile code assuming that floating-point operations cannot generate user-visible traps. Setting this option may allow faster code if one relies on "non-stop" IEEE arithmetic, for example.

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

  The default is '`-ftrapping-math`'.

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

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

Not all of the optimizations performed by GCC have '`-f`' options to control them.

`-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.

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

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

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

`-frerun-loop-opt`
: Run the loop optimizer twice.

`-fgcse`
: Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

  Note: When compiling a program using computed gotos, a GCC extension, you may get better runtime performance if you disable the global common subexpression elmination pass by adding '`-fno-gcse`' to the command line.

`-fgcse-lm`
: When '`-fgcse-lm`' is enabled, global common subexpression elimination will attempt to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.

`-fgcse-sm`
: When '`-fgcse-sm`' is enabled, A store motion pass is run after global common subexpression elimination. This pass will attempt to move stores out of loops. When used in conjunction with '`-fgcse-lm`', loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.

`-fdelete-null-pointer-checks`
: Use global dataflow analysis to identify and eliminate useless checks for null pointers. The compiler assumes that dereferencing a null pointer would have halted the program. If a pointer is checked after it has already been dereferenced, it cannot be null.

  In some environments, this assumption is not true, and programs can safely dereference null pointers. Use '`-fno-delete-null-pointer-checks`' to disable this optimization for programs which depend on that behavior.

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

`-foptimize-register-move`
`-fregmove`

Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. This is especially helpful on machines with two-operand instructions. GCC enables this optimization by default with '`-O2`' or higher.

Note '`-fregmove`' and '`-foptimize-register-move`' are the same optimization.

`-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.

`-ffunction-sections`
`-fdata-sections`

Place each function or data item into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section's name in the output file.

Use these options on systems where the linker can perform optimizations to improve locality of reference in the instruction space. HPPA processors running HP-UX and Sparc processors running Solaris 2 have linkers with such optimizations. Other systems using the ELF object format as well as AIX may have these optimizations in the future.

Only use these options when there are significant benefits from doing so. When you specify these options, the assembler and linker will create larger object and executable files and will also be slower. You will not be able to use `gprof` on all systems if you specify this option and you may have problems with debugging if you specify both this option and '`-g`'.

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

For all machines, optimization level 2 and higher enables this flag by default.

`-funroll-loops`
: Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. '`-funroll-loops`' implies both '`-fstrength-reduce`' and '`-frerun-cse-after-loop`'. This option makes code larger, and may or may not make it run faster.

`-funroll-all-loops`
: Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. '`-funroll-all-loops`' implies the same options as '`-funroll-loops`',

`-fprefetch-loop-arrays`
: If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

`-fmove-all-movables`
: Forces all invariant computations in loops to be moved outside the loop.

`-freduce-all-givs`
: Forces all general-induction variables in loops to be strength-reduced.

  Note: When compiling programs written in Fortran, '`-fmove-all-movables`' and '`-freduce-all-givs`' are enabled by default when you use the optimizer.

  These options may generate better or worse code; results are highly dependent on the structure of loops within the source code.

  These two options are intended to be removed someday, once they have helped determine the efficacy of various approaches to improving loop optimizations.

  Please let us (`gcc@gcc.gnu.org` and `fortran@gnu.org`) know how use of these options affects the performance of your production code. We're very interested in code that runs *slower* when these options are *enabled*.

`-fno-peephole`
`-fno-peephole2`
: Disable any machine-specific peephole optimizations. The difference between '`-fno-peephole`' and '`-fno-peephole2`' is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.

`-fbranch-probabilities`
: After running a program compiled with '`-fprofile-arcs`' (see Section 3.9 [Options for Debugging Your Program or `gcc`], page 43), you can compile it a second time using '`-fbranch-probabilities`', to improve optimizations based on the number of times each branch was taken. When the program compiled with '`-fprofile-arcs`' exits it saves arc execution counts to a file called '*sourcename*`.da`' for each source file The information in this data file is very dependent on the structure of the generated code, so you must use the same source code and the same optimization options for both compilations.

  With '`-fbranch-probabilities`', GCC puts a '`REG_EXEC_COUNT`' note on the first instruction of each basic block, and a '`REG_BR_PROB`' note on each '`JUMP_INSN`' and '`CALL_INSN`'. These can be used to improve optimization.

Currently, they are only used in one place: in '`reorg.c`', instead of guessing which path a branch is mostly to take, the '`REG_BR_PROB`' values are used to exactly determine which path is taken more often.

`-fno-guess-branch-probability`
Do not guess branch probabilities using a randomized model.

Sometimes gcc will opt to use a randomized model to guess branch probabilities, when none are available from either profiling feedback ('`-fprofile-arcs`') or '`__builtin_expect`'. This means that different runs of the compiler on the same program may produce different object code.

In a hard real-time system, people don't want different runs of the compiler to produce code that has different behavior; minimizing non-determinism is of paramount import. This switch allows users to reduce non-determinism, possibly at the expense of inferior optimization.

`-fstrict-aliasing`
Allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C (and C++), this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.

Pay special attention to code like this:

```
union a_union {
  int i;
  double d;
};

int f() {
  a_union t;
  t.d = 3.0;
  return t.i;
}
```

The practice of reading from a different union member than the one most recently written to (called "type-punning") is common. Even with '`-fstrict-aliasing`', type-punning is allowed, provided the memory is accessed through the union type. So, the code above will work as expected. However, this code might not:

```
int f() {
  a_union t;
  int* ip;
  t.d = 3.0;
  ip = &t.i;
  return *ip;
}
```

Every language that wishes to perform language-specific alias analysis should define a function that computes, given an `tree` node, an alias set for the node.

Nodes in different alias sets are not allowed to alias. For an example, see the C front-end function `c_get_alias_set`.

`-falign-functions`
`-falign-functions=n`
: Align the start of functions to the next power-of-two greater than *n*, skipping up to *n* bytes. For instance, '`-falign-functions=32`' aligns functions to the next 32-byte boundary, but '`-falign-functions=24`' would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less.

'`-fno-align-functions`' and '`-falign-functions=1`' are equivalent and mean that functions will not be aligned.

Some assemblers only support this flag when *n* is a power of two; in that case, it is rounded up.

If *n* is not specified, use a machine-dependent default.

`-falign-labels`
`-falign-labels=n`
: Align all branch targets to a power-of-two boundary, skipping up to *n* bytes like '`-falign-functions`'. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code.

If '`-falign-loops`' or '`-falign-jumps`' are applicable and are greater than this value, then their values are used instead.

If *n* is not specified, use a machine-dependent default which is very likely to be '`1`', meaning no alignment.

`-falign-loops`
`-falign-loops=n`
: Align loops to a power-of-two boundary, skipping up to *n* bytes like '`-falign-functions`'. The hope is that the loop will be executed many times, which will make up for any execution of the dummy operations.

If *n* is not specified, use a machine-dependent default.

`-falign-jumps`
`-falign-jumps=n`
: Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to *n* bytes like '`-falign-functions`'. In this case, no dummy operations need be executed.

If *n* is not specified, use a machine-dependent default.

`-fssa`
: Perform optimizations in static single assignment form. Each function's flow graph is translated into SSA form, optimizations are performed, and the flow graph is translated back from SSA form. Users should not specify this option, since it is not yet ready for production use.

`-fssa-ccp`
: Perform Sparse Conditional Constant Propagation in SSA form. Requires '`-fssa`'. Like '`-fssa`', this is an experimental feature.

`-fssa-dce`
: Perform aggressive dead-code elimination in SSA form. Requires '`-fssa`'. Like '`-fssa`', this is an experimental feature.

`-fsingle-precision-constant`
: Treat floating point constant as single precision constant instead of implicitly converting it to double precision constant.

`-frename-registers`
: Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization will most benefit processors with lots of registers. It can, however, make debugging impossible, since variables will no longer stay in a "home register".

`-fno-cprop-registers`
: After register allocation and post-register allocation instruction splitting, we perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.

`--param` *name*=*value*
: In some places, GCC uses various constants to control the amount of optimization that is done. For example, GCC will not inline functions that contain more that a certain number of instructions. You can control some of these constants on the command-line using the '`--param`' option.

  In each case, the *value* is an integer. The allowable choices for *name* are given in the following table:

  `max-delay-slot-insn-search`
  : The maximum number of instructions to consider when looking for an instruction to fill a delay slot. If more than this arbitrary number of instructions is searched, the time savings from filling the delay slot will be minimal so stop searching. Increasing values mean more aggressive optimization, making the compile time increase with probably small improvement in executable run time.

  `max-delay-slot-live-search`
  : When trying to fill delay slots, the maximum number of instructions to consider when searching for a block with valid live register information. Increasing this arbitrarily chosen value means more aggressive optimization, increasing the compile time. This parameter should be removed when the delay slot code is rewritten to maintain the control-flow graph.

  `max-gcse-memory`
  : The approximate maximum amount of memory that will be allocated in order to perform the global common subexpression elimination optimization. If more memory than specified is required, the optimization will not be done.

  `max-gcse-passes`
  : The maximum number of passes of GCSE to run.

`max-pending-list-length`
: The maximum number of pending dependencies scheduling will allow before flushing the current state and starting over. Large functions with few branches or calls can create excessively large lists which needlessly consume memory and resources.

`max-inline-insns`
: If an function contains more than this many instructions, it will not be inlined. This option is precisely equivalent to '`-finline-limit`'.

3.11 Options Controlling the Preprocessor

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

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

You can use '`-Wp,`*option*' to bypass the compiler driver and pass *option* directly through to the preprocessor. If *option* contains commas, it is split into multiple options at the commas. However, many options are modified, translated or interpreted by the compiler driver before being passed to the preprocessor, and '`-Wp`' forcibly bypasses this phase. The preprocessor's direct interface is undocumented and subject to change, so whenever possible you should avoid using '`-Wp`' and let the driver handle the options instead.

`-D` *name*
: Predefine *name* as a macro, with definition `1`.

`-D` *name*=*definition*
: Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

: If you wish to define a function-like macro on the command line, write its argument list with surrounding parentheses before the equals sign (if any). Parentheses are meaningful to most shells, so you will need to quote the option. With `sh` and `csh`, '`-D'`*name*`(`*args...*`)=`*definition*`''` works.

: '`-D`' and '`-U`' options are processed in the order they are given on the command line. All '`-imacros` *file*' and '`-include` *file*' options are processed after all '`-D`' and '`-U`' options.

`-U` *name*
: Cancel any previous definition of *name*, either built in or provided with a '`-D`' option.

`-undef`
: Do not predefine any system-specific macros. The common predefined macros remain defined.

`-I` *dir*
: Add the directory *dir* to the list of directories to be searched for header files. Directories named by '`-I`' are searched before the standard system include directories.

It is dangerous to specify a standard system include directory in an '-I' option. This defeats the special treatment of system headers . It can also defeat the repairs to buggy system headers which GCC makes when it is installed.

-o *file*
: Write output to *file*. This is the same as specifying *file* as the second non-option argument to cpp. gcc has a different interpretation of a second non-option argument, so you must use '-o' to specify the output file.

-Wall
: Turns on all optional warnings which are desirable for normal code. At present this is '-Wcomment' and '-Wtrigraphs'. Note that many of the preprocessor's warnings are on by default and have no options to control them.

-Wcomment
-Wcomments
: Warn whenever a comment-start sequence '/*' appears in a '/*' comment, or whenever a backslash-newline appears in a '//' comment. (Both forms have the same effect.)

-Wtrigraphs
: Warn if any trigraphs are encountered. This option used to take effect only if '-trigraphs' was also specified, but now works independently. Warnings are not given for trigraphs within comments, as they do not affect the meaning of the program.

-Wtraditional
: Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and problematic constructs which should be avoided.

-Wimport
: Warn the first time '#import' is used.

-Wundef
: Warn whenever an identifier which is not a macro is encountered in an '#if' directive, outside of 'defined'. Such identifiers are replaced with zero.

-Werror
: Make all warnings into hard errors. Source code which triggers warnings will be rejected.

-Wsystem-headers
: Issue warnings for code in system headers. These are normally unhelpful in finding bugs in your own code, therefore suppressed. If you are responsible for the system library, you may want to see them.

-w
: Suppress all warnings, including those which GNU CPP issues by default.

-pedantic
: Issue all the mandatory diagnostics listed in the C standard. Some of them are left out by default, since they trigger frequently on harmless code.

-pedantic-errors
: Issue all the mandatory diagnostics, and make all mandatory diagnostics into errors. This includes mandatory diagnostics that GCC issues without '-pedantic' but treats as warnings.

-M
: Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs

one `make` rule containing the object file name for that source file, a colon, and the names of all the included files, including those coming from '`-include`' or '`-imacros`' command line options.

Unless specified explicitly (with '`-MT`' or '`-MQ`'), the object file name consists of the basename of the source file with any suffix replaced with object file suffix. If there are many included files then the rule is split into several lines using '`\`'-newline. The rule has no commands.

This option does not suppress the preprocessor's debug output, such as '`-dM`'. To avoid mixing such debug output with the dependency rules you should explicitly specify the dependency output file with '`-MF`', or use an environment variable like `DEPENDENCIES_OUTPUT` (see [DEPENDENCIES_OUTPUT], page 147). Debug output will still be sent to the regular output stream as normal.

Passing '`-M`' to the driver implies '`-E`'.

`-MM` Like '`-M`' but do not mention header files that are found in system header directories, nor header files that are included, directly or indirectly, from such a header.

This implies that the choice of angle brackets or double quotes in an '`#include`' directive does not in itself determine whether that header will appear in '`-MM`' dependency output. This is a slight change in semantics from GCC versions 3.0 and earlier.

`-MF` *file* When used with '`-M`' or '`-MM`', specifies a file to write the dependencies to. If no '`-MF`' switch is given the preprocessor sends the rules to the same place it would have sent preprocessed output.

When used with the driver options '`-MD`' or '`-MMD`', '`-MF`' overrides the default dependency output file.

`-MG` When used with '`-M`' or '`-MM`', '`-MG`' says to treat missing header files as generated files and assume they live in the same directory as the source file. It suppresses preprocessed output, as a missing header file is ordinarily an error.

This feature is used in automatic updating of makefiles.

`-MP` This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors `make` gives if you remove header files without updating the '`Makefile`' to match.

This is typical output:

```
test.o: test.c test.h

test.h:
```

`-MT` *target*

Change the target of the rule emitted by dependency generation. By default CPP takes the name of the main input file, including any path, deletes any file suffix such as '`.c`', and appends the platform's usual object suffix. The result is the target.

An '`-MT`' option will set the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to '`-MT`', or use multiple '`-MT`' options.

For example, '`-MT '$(objpfx)foo.o'`' might give

```
$(objpfx)foo.o: foo.c
```

`-MQ` *target*

Same as '`-MT`', but it quotes any characters which are special to Make. '`-MQ '$(objpfx)foo.o'`' gives

```
$$(objpfx)foo.o: foo.c
```

The default target is automatically quoted, as if it were given with '`-MQ`'.

`-MD` '`-MD`' is equivalent to '`-M -MF` *file*', except that '`-E`' is not implied. The driver determines *file* based on whether an '`-o`' option is given. If it is, the driver uses its argument but with a suffix of '`.d`', otherwise it take the basename of the input file and applies a '`.d`' suffix.

If '`-MD`' is used in conjunction with '`-E`', any '`-o`' switch is understood to specify the dependency output file (but see [-MF], page 63), but if used without '`-E`', each '`-o`' is understood to specify a target object file.

Since '`-E`' is not implied, '`-MD`' can be used to generate a dependency output file as a side-effect of the compilation process.

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

`-x c`
`-x c++`
`-x objective-c`
`-x assembler-with-cpp`

Specify the source language: C, C++, Objective-C, or assembly. This has nothing to do with standards conformance or extensions; it merely selects which base syntax to expect. If you give none of these options, cpp will deduce the language from the extension of the source file: '`.c`', '`.cc`', '`.m`', or '`.S`'. Some other common extensions for C++ and assembly are also recognized. If cpp does not recognize the extension, it will treat the file as C; this is the most generic mode.

Note: Previous versions of cpp accepted a '`-lang`' option which selected both the language and the standards conformance level. This option has been removed, because it conflicts with the '`-l`' option.

`-std=`*standard*
`-ansi` Specify the standard to which the code should conform. Currently cpp only knows about the standards for C; other language standards will be added in the future.

standard may be one of:

`iso9899:1990`
`c89` The ISO C standard from 1990. '`c89`' is the customary shorthand for this version of the standard.

The '`-ansi`' option is equivalent to '`-std=c89`'.

`iso9899:199409`
: The 1990 C standard, as amended in 1994.

`iso9899:1999`
`c99`
`iso9899:199x`
`c9x`
: The revised ISO C standard, published in December 1999. Before publication, this was known as C9X.

`gnu89`
: The 1990 C standard plus GNU extensions. This is the default.

`gnu99`
`gnu9x`
: The 1999 C standard plus GNU extensions.

`-I-`
: Split the include path. Any directories specified with '`-I`' options before '`-I-`' are searched only for headers requested with `#include "`*file*`"`; they are not searched for `#include <`*file*`>`. If additional directories are specified with '`-I`' options after the '`-I-`', those directories are searched for all '`#include`' directives.

  In addition, '`-I-`' inhibits the use of the directory of the current file directory as the first search directory for `#include "`*file*`"`.

`-nostdinc`
: Do not search the standard system directories for header files. Only the directories you have specified with '`-I`' options (and the directory of the current file, if appropriate) are searched.

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

`-include` *file*
: Process *file* as if `#include "file"` appeared as the first line of the primary source file. However, the first directory searched for *file* is the preprocessor's working directory *instead of* the directory containing the main source file. If not found there, it is searched for in the remainder of the `#include "..."` search chain as normal.

  If multiple '`-include`' options are given, the files are included in the order they appear on the command line.

`-imacros` *file*
: Exactly like '`-include`', except that any output produced by scanning *file* is thrown away. Macros it defines remain defined. This allows you to acquire all the macros from a header without also processing its declarations.

  All files specified by '`-imacros`' are processed before all files specified by '`-include`'.

`-idirafter` *dir*
: Search *dir* for header files, but do it *after* all directories specified with '`-I`' and the standard system directories have been exhausted. *dir* is treated as a system include directory.

`-iprefix` *prefix*

Specify *prefix* as the prefix for subsequent '`-iwithprefix`' options. If the prefix represents a directory, you should include the final '`/`'.

`-iwithprefix` *dir*
`-iwithprefixbefore` *dir*

Append *dir* to the prefix specified previously with '`-iprefix`', and add the resulting directory to the include search path. '`-iwithprefixbefore`' puts it in the same place '`-I`' would; '`-iwithprefix`' puts it where '`-idirafter`' would.

Use of these options is discouraged.

`-isystem` *dir*

Search *dir* for header files, after all directories specified by '`-I`' but before the standard system directories. Mark it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

`-fpreprocessed`

Indicate to the preprocessor that the input file has already been preprocessed. This suppresses things like macro expansion, trigraph conversion, escaped newline splicing, and processing of most directives. The preprocessor still recognizes and removes comments, so that you can pass a file preprocessed with '`-C`' to the compiler without problems. In this mode the integrated preprocessor is little more than a tokenizer for the front ends.

'`-fpreprocessed`' is implicit if the input file has one of the extensions '`.i`', '`.ii`' or '`.mi`'. These are the extensions that GCC uses for preprocessed files created by '`-save-temps`'.

`-ftabstop=`*width*

Set the distance between tab stops. This helps the preprocessor report correct column numbers in warnings or errors, even if tabs appear on the line. If the value is less than 1 or greater than 100, the option is ignored. The default is 8.

`-fno-show-column`

Do not print column numbers in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as `dejagnu`.

`-A` *predicate*=*answer*

Make an assertion with the predicate *predicate* and answer *answer*. This form is preferred to the older form '`-A` *predicate*`(`*answer*`)`', which is still supported, because it does not use shell special characters.

`-A -`*predicate*=*answer*

Cancel an assertion with the predicate *predicate* and answer *answer*.

`-A-` Cancel all predefined assertions and all assertions preceding it on the command line. Also, undefine all predefined macros and all macros preceding it on the command line. (This is a historical wart and may change in the future.)

`-dCHARS` *CHARS* is a sequence of one or more of the following characters, and must not be preceded by a space. Other characters are interpreted by the compiler proper, or reserved for future versions of GCC, and so are silently ignored. If you specify characters whose behavior conflicts, the result is undefined.

'M' Instead of the normal output, generate a list of '#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor. Assuming you have no file 'foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

will show all the predefined macros.

'D' Like 'M' except in two respects: it does *not* include the predefined macros, and it outputs *both* the '#define' directives and the result of preprocessing. Both kinds of output go to the standard output file.

'N' Like 'D', but emit only the macro names, not their expansions.

'I' Output '#include' directives in addition to the result of preprocessing.

-P Inhibit generation of linemarkers in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code, and will be sent to a program which might be confused by the linemarkers.

-C Do not discard comments. All comments are passed through to the output file, except for comments in processed directives, which are deleted along with the directive.

You should be prepared for side effects when using '-C'; it causes the preprocessor to treat comments as tokens in their own right. For example, comments appearing at the start of what would be a directive line have the effect of turning that line into an ordinary source line, since the first token on the line is no longer a '#'.

-gcc Define the macros __GNUC__, __GNUC_MINOR__ and __GNUC_PATCHLEVEL__. These are defined automatically when you use `gcc -E`; you can turn them off in that case with '-no-gcc'.

-traditional
Try to imitate the behavior of old-fashioned C, as opposed to ISO C.

-trigraphs
Process trigraph sequences. These are three-character sequences, all starting with '??', that are defined by ISO C to stand for single characters. For example, '??/' stands for '\', so ''??/n'' is a character constant for a newline. By default, GCC ignores trigraphs, but in standard-conforming modes it converts them. See the '-std' and '-ansi' options.

The nine trigraphs and their replacements are

```
Trigraph:       ??(  ??)  ??<  ??>  ??=  ??/  ??'  ??!  ??-
Replacement:      [    ]    {    }    #    \    ^    |    ~
```

`-remap` Enable special code to work around file systems which only permit very short file names, such as MS-DOS.

`-$` Forbid the use of '`$`' in identifiers. The C standard allows implementations to define extra characters that can appear in identifiers. By default GNU CPP permits '`$`', a common extension.

`-h`
`--help`
`--target-help`
Print text describing all the command line options instead of preprocessing anything.

`-v` Verbose mode. Print out GNU CPP's version number at the beginning of execution, and report the final form of the include path.

`-H` Print the name of each header file used, in addition to other normal activities. Each name is indented to show how deep in the '`#include`' stack it is.

`-version`
`--version`
Print out GNU CPP's version number. With one dash, proceed to preprocess as normal. With two dashes, exit immediately.

3.12 Passing Options to the Assembler

You can pass options to the assembler.

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

3.13 Options for Linking

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

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

`-c`
`-S`
`-E` If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See Section 3.2 [Overall Options], page 16.

`-l`*library*
`-l` *library* Search the library named *library* when linking. (The second alternative with the library as a separate argument is only for POSIX compliance and is not recommended.)

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

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

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

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

`-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 system libraries are used normally, unless '`-nostdlib`' or '`-nodefaultlibs`' is used.

`-nodefaultlibs`
Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files are used normally, unless '`-nostartfiles`' is used. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ISO C) environments or to bcopy and bzero for BSD environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

`-nostdlib`
Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ISO C) environments or to bcopy and bzero for BSD environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

One of the standard libraries bypassed by '`-nostdlib`' and '`-nodefaultlibs`' is '`libgcc.a`', a library of internal subroutines that GCC uses to overcome shortcomings of particular machines, or special needs for some languages. (See section "Interfacing to GCC Output" in *GNU Compiler Collection (GCC) Internals*, for more discussion of '`libgcc.a`'.) In most cases, you need '`libgcc.a`' even when you want to avoid other standard libraries. In other words, when you specify '`-nostdlib`' or '`-nodefaultlibs`' you should usually specify '`-lgcc`' as well. This ensures that you have no unresolved references to internal GCC library subroutines. (For example, '`__main`', used to ensure C++ constructors will be called; see section "`collect2`" in *GNU Compiler Collection (GCC) Internals*.)

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

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

`-shared` Produce a shared object which can then be linked with other objects to form an executable. Not all systems support this option. For predictable results, you must also specify the same set of options that were used to generate code ('`-fpic`', '`-fPIC`', or model suboptions) when you specify this option.[1]

`-shared-libgcc`
`-static-libgcc`

On systems that provide '`libgcc`' as a shared library, these options force the use of either the shared or static version respectively. If no shared version of '`libgcc`' was built when the compiler was configured, these options have no effect.

There are several situations in which an application should use the shared '`libgcc`' instead of the static version. The most common of these is when the application wishes to throw and catch exceptions across different shared libraries. In that case, each of the libraries as well as the application itself should use the shared '`libgcc`'.

Therefore, the G++ and GCJ drivers automatically add '`-shared-libgcc`' whenever you build a shared library or a main executable, because C++ and Java programs typically use exceptions, so this is the right thing to do.

If, instead, you use the GCC driver to create shared libraries, you may find that they will not always be linked with the shared '`libgcc`'. If GCC finds, at its configuration time, that you have a GNU linker that does not support option '`--eh-frame-hdr`', it will link the shared version of '`libgcc`' into shared libraries by default. Otherwise, it will take advantage of the linker and optimize away the linking with the shared version of '`libgcc`', linking with the static version of libgcc by default. This allows exceptions to propagate through such shared libraries, without incurring relocation costs at library load time.

However, if a library or main executable is supposed to throw or catch exceptions, you must link it using the G++ or GCJ driver, as appropriate for the languages used in the program, or using the option '`-shared-libgcc`', such that it is linked with the shared '`libgcc`'.

`-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.

[1] On some systems, '`gcc -shared`' needs to build supplementary stub code for constructors to work. On multi-libbed systems, '`gcc -shared`' must select the correct support libraries to link against. Failing to supply the correct flags may lead to subtle defects. Supplying them in cases where they are not necessary is innocuous.

`-Xlinker` *option*
: Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GCC 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.

3.14 Options for Directory Search

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

`-I`*dir*
: Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. However, you should not use this option to add directories that contain vendor-supplied system header files (use '`-isystem`' for that). If you use more than one '`-I`' option, the directories are scanned in left-to-right order; the standard system directories come after.

  If a standard system include directory, or a directory specified with '`-isystem`', is also specified with '`-I`', it will be searched only in the position requested by '`-I`'. Also, it will not be considered a system include directory. If that directory really does contain system headers, there is a good chance that they will break. For instance, if GCC's installation procedure edited the headers in '`/usr/include`' to fix bugs, '`-I/usr/include`' will cause the original, buggy headers to be found instead of the corrected ones. GCC will issue a warning when a system include directory is hidden in this way.

`-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, include files, and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms '`cpp`', '`cc1`', '`as`' and '`ld`'. It tries *prefix* as a prefix for each program it tries to run, both with and without '*machine*/*version*/' (see Section 3.16 [Target Options], page 79).

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

The compiler will check to see if the path provided by the '`-B`' refers to a directory, and if necessary it will add a directory separator character at the end of the path.

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

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

Another way to specify a prefix much like the '`-B`' prefix is to use the environment variable `GCC_EXEC_PREFIX`. See Section 3.19 [Environment Variables], page 145.

As a special kludge, if the path provided by '`-B`' is '`[dir/]stage`*N*`/`', where *N* is a number in the range 0 to 9, then it will be replaced by '`[dir/]include`'. This is to help with boot-strapping the compiler.

`-specs=`*file*

Process *file* after the compiler reads in the standard '`specs`' file, in order to override the defaults that the '`gcc`' driver program uses when determining what switches to pass to '`cc1`', '`cc1plus`', '`as`', '`ld`', etc. More than one '`-specs=`*file*' can be specified on the command line, and they are processed in order, from left to right.

3.15 Specifying subprocesses and the switches to pass to them

`gcc` is a driver program. It performs its job by invoking a sequence of other programs to do the work of compiling, assembling and linking. GCC interprets its command-line parameters and uses these to deduce which programs it should invoke, and which command-line options it ought to place on their command lines. This behavior is controlled by *spec strings*. In most cases there is one spec string for each program that GCC can invoke, but a few programs have multiple spec strings to control their behavior. The spec strings built into GCC can be overridden by using the '`-specs=`' command-line switch to specify a spec file.

Spec files are plaintext files that are used to construct spec strings. They consist of a sequence of directives separated by blank lines. The type of directive is determined by the first non-whitespace character on the line and it can be one of the following:

`%`*command*
: Issues a *command* to the spec file processor. The commands that can appear here are:

  `%include` <*file*>
  : Search for *file* and insert its text at the current point in the specs file.

  `%include_noerr` <*file*>
  : Just like '`%include`', but do not generate an error message if the include file cannot be found.

  `%rename` *old_name new_name*
  : Rename the spec string *old_name* to *new_name*.

`*[`*spec_name*`]:`
: This tells the compiler to create, override or delete the named spec string. All lines after this directive up to the next directive or blank line are considered to be the text for the spec string. If this results in an empty string then the spec will be deleted. (Or, if the spec did not exist, then nothing will happened.) Otherwise, if the spec does not currently exist a new spec will be created. If the spec does exist then its contents will be overridden by the text of this directive, unless the first character of that text is the '`+`' character, in which case the text will be appended to the spec.

`[`*suffix*`]:`
: Creates a new '`[`*suffix*`] spec`' pair. All lines after this directive and up to the next directive or blank line are considered to make up the spec string for the indicated suffix. When the compiler encounters an input file with the named suffix, it will processes the spec string in order to work out how to compile that file. For example:

  ```
  .ZZ:
  z-compile -input %i
  ```

  This says that any input file whose name ends in '`.ZZ`' should be passed to the program '`z-compile`', which should be invoked with the command-line switch '`-input`' and with the result of performing the '`%i`' substitution. (See below.)

As an alternative to providing a spec string, the text that follows a suffix directive can be one of the following:

`@`*language* This says that the suffix is an alias for a known *language*. This is similar to using the '`-x`' command-line switch to GCC to specify a language explicitly. For example:

```
.ZZ:
@c++
```

Says that .ZZ files are, in fact, C++ source files.

`#`*name* This causes an error messages saying:

```
name compiler not installed on this system.
```

GCC already has an extensive list of suffixes built into it. This directive will add an entry to the end of the list of suffixes, but since the list is searched from the end backwards, it is effectively possible to override earlier entries using this technique.

GCC has the following spec strings built into it. Spec files can override these strings or create their own. Note that individual targets can also add their own spec strings to this list.

```
asm          Options to pass to the assembler
asm_final    Options to pass to the assembler post-processor
cpp          Options to pass to the C preprocessor
cc1          Options to pass to the C compiler
cc1plus      Options to pass to the C++ compiler
endfile      Object files to include at the end of the link
link         Options to pass to the linker
lib          Libraries to include on the command line to the linker
libgcc       Decides which GCC support library to pass to the linker
linker       Sets the name of the linker
predefines   Defines to be passed to the C preprocessor
signed_char  Defines to pass to CPP to say whether char is signed
             by default
startfile    Object files to include at the start of the link
```

Here is a small example of a spec file:

```
%rename lib                 old_lib

*lib:
--start-group -lgcc -lc -leval1 --end-group %(old_lib)
```

This example renames the spec called '`lib`' to '`old_lib`' and then overrides the previous definition of '`lib`' with a new one. The new definition adds in some extra command-line options before including the text of the old definition.

Spec strings are a list of command-line options to be passed to their corresponding program. In addition, the spec strings can contain '`%`'-prefixed sequences to substitute variable text or to conditionally insert text into the command line. Using these constructs it is possible to generate quite complex command lines.

Here is a table of all defined '%'-sequences for spec strings. Note that spaces are not generated automatically around the results of expanding these sequences. Therefore you can concatenate them together or combine them with constant text in a single argument.

`%%` Substitute one '%' into the program name or argument.

`%i` Substitute the name of the input file being processed.

`%b` Substitute the basename of the input file being processed. This is the substring up to (and not including) the last period and not including the directory.

`%B` This is the same as '%b', but include the file suffix (text after the last period).

`%d` Marks the argument containing or following the '%d' as a temporary file name, so that that file will be deleted if GCC exits successfully. Unlike '%g', this contributes no text to the argument.

`%g`*suffix* Substitute a file name that has suffix *suffix* and is chosen once per compilation, and mark the argument in the same way as '%d'. To reduce exposure to denial-of-service attacks, the file name is now chosen in a way that is hard to predict even when previously chosen file names are known. For example, '%g.s ... %g.o ... %g.s' might turn into 'ccUVUUAU.s ccXYAXZ12.o ccUVUUAU.s'. *suffix* matches the regexp '[.A-Za-z]*' or the special string '%O', which is treated exactly as if '%O' had been preprocessed. Previously, '%g' was simply substituted with a file name chosen once per compilation, without regard to any appended suffix (which was therefore treated just like ordinary text), making such attacks more likely to succeed.

`%u`*suffix* Like '%g', but generates a new temporary file name even if '%u*suffix*' was already seen.

`%U`*suffix* Substitutes the last file name generated with '%u*suffix*', generating a new one if there is no such last file name. In the absence of any '%u*suffix*', this is just like '%g*suffix*', except they don't share the same suffix *space*, so '%g.s ... %U.s ... %g.s ... %U.s' would involve the generation of two distinct file names, one for each '%g.s' and another for each '%U.s'. Previously, '%U' was simply substituted with a file name chosen for the previous '%u', without regard to any appended suffix.

`%j`*SUFFIX*
Substitutes the name of the HOST_BIT_BUCKET, if any, and if it is writable, and if save-temps is off; otherwise, substitute the name of a temporary file, just like '%u'. This temporary file is not meant for communication between processes, but rather as a junk disposal mechanism.

`%.`*SUFFIX*
Substitutes *.SUFFIX* for the suffixes of a matched switch's args when it is subsequently output with '%*'. *SUFFIX* is terminated by the next space or %.

`%w` Marks the argument containing or following the '%w' as the designated output file of this compilation. This puts the argument into the sequence of arguments that '%o' will substitute later.

`%o` Substitutes the names of all the output files, with spaces automatically placed around them. You should write spaces around the '`%o`' as well or the results are undefined. '`%o`' is for use in the specs for running the linker. Input files whose names have no recognized suffix are not compiled at all, but they are included among the output files, so they will be linked.

`%O` Substitutes the suffix for object files. Note that this is handled specially when it immediately follows '`%g, %u, or %U`', because of the need for those to form complete file names. The handling is such that '`%O`' is treated exactly as if it had already been substituted, except that '`%g, %u, and %U`' do not currently support additional *suffix* characters following '`%O`' as they would following, for example, '`.o`'.

`%p` Substitutes the standard macro predefinitions for the current target machine. Use this when running `cpp`.

`%P` Like '`%p`', but puts '`__`' before and after the name of each predefined macro, except for macros that start with '`__`' or with '`_L`', where *L* is an uppercase letter. This is for ISO C.

`%I` Substitute a '`-iprefix`' option made from `GCC_EXEC_PREFIX`.

`%s` Current argument is the name of a library or startup file of some sort. Search for that file in a standard list of directories and substitute the full name found.

`%e`*str* Print *str* as an error message. *str* is terminated by a newline. Use this when inconsistent options are detected.

`%|` Output '`-`' if the input for the current command is coming from a pipe.

`%(`*name*`)` Substitute the contents of spec string *name* at this point.

`%[`*name*`]` Like '`%(...)`' but put '`__`' around '`-D`' arguments.

`%x{`*option*`}`
Accumulate an option for '`%X`'.

`%X` Output the accumulated linker options specified by '`-Wl`' or a '`%x`' spec string.

`%Y` Output the accumulated assembler options specified by '`-Wa`'.

`%Z` Output the accumulated preprocessor options specified by '`-Wp`'.

`%v1` Substitute the major version number of GCC. (For version 2.9.5, this is 2.)

`%v2` Substitute the minor version number of GCC. (For version 2.9.5, this is 9.)

`%v3` Substitute the patch level number of GCC. (For version 2.9.5, this is 5.)

`%a` Process the `asm` spec. This is used to compute the switches to be passed to the assembler.

`%A` Process the `asm_final` spec. This is a spec string for passing switches to an assembler post-processor, if such a program is needed.

`%l` Process the `link` spec. This is the spec for computing the command line passed to the linker. Typically it will make use of the '`%L %G %S %D and %E`' sequences.

`%D` Dump out a '-L' option for each directory that GCC believes might contain startup files. If the target supports multilibs then the current multilib directory will be prepended to each of these paths.

`%M` Output the multilib directory with directory separators replaced with '_'. If multilib directories are not set, or the multilib directory is '.' then this option emits nothing.

`%L` Process the `lib` spec. This is a spec string for deciding which libraries should be included on the command line to the linker.

`%G` Process the `libgcc` spec. This is a spec string for deciding which GCC support library should be included on the command line to the linker.

`%S` Process the `startfile` spec. This is a spec for deciding which object files should be the first ones passed to the linker. Typically this might be a file named 'crt0.o'.

`%E` Process the `endfile` spec. This is a spec string that specifies the last object files that will be passed to the linker.

`%C` Process the `cpp` spec. This is used to construct the arguments to be passed to the C preprocessor.

`%c` Process the `signed_char` spec. This is intended to be used to tell cpp whether a char is signed. It typically has the definition:

```
%{funsigned-char:-D__CHAR_UNSIGNED__}
```

`%1` Process the `cc1` spec. This is used to construct the options to be passed to the actual C compiler ('cc1').

`%2` Process the `cc1plus` spec. This is used to construct the options to be passed to the actual C++ compiler ('cc1plus').

`%*` Substitute the variable part of a matched option. See below. Note that each comma in the substituted string is replaced by a single space.

`%{S}` Substitutes the `-S` switch, if that switch was given to GCC. If that switch was not specified, this substitutes nothing. Note that the leading dash is omitted when specifying this option, and it is automatically inserted if the substitution is performed. Thus the spec string '%{foo}' would match the command-line option '-foo' and would output the command line option '-foo'.

`%W{S}` Like %{S} but mark last argument supplied within as a file to be deleted on failure.

`%{S*}` Substitutes all the switches specified to GCC whose names start with `-S`, but which also take an argument. This is used for switches like '-o', '-D', '-I', etc. GCC considers '-o foo' as being one switch whose names starts with 'o'. %{o*} would substitute this text, including the space. Thus two arguments would be generated.

`%{^S*}` Like %{S*}, but don't put a blank between a switch and its argument. Thus %{^o*} would only generate one argument, not two.

`%{S*&T*}` Like `%{S*}`, but preserve order of `S` and `T` options (the order of `S` and `T` in the spec is not significant). There can be any number of ampersand-separated variables; for each the wild card is optional. Useful for CPP as '`%{D*&U*&A*}`'.

`%{<S}` Remove all occurrences of `-S` from the command line. Note—this command is position dependent. '`%`' commands in the spec string before this option will see `-S`, '`%`' commands in the spec string after this option will not.

`%{S*:X}` Substitutes `X` if one or more switches whose names start with `-S` are specified to GCC. Note that the tail part of the `-S` option (i.e. the part matched by the '`*`') will be substituted for each occurrence of '`%*`' within `X`.

`%{S:X}` Substitutes `X`, but only if the '`-S`' switch was given to GCC.

`%{!S:X}` Substitutes `X`, but only if the '`-S`' switch was *not* given to GCC.

`%{|S:X}` Like `%{S:X}`, but if no `S` switch, substitute '`-`'.

`%{|!S:X}` Like `%{!S:X}`, but if there is an `S` switch, substitute '`-`'.

`%{.S:X}` Substitutes `X`, but only if processing a file with suffix `S`.

`%{!.S:X}` Substitutes `X`, but only if *not* processing a file with suffix `S`.

`%{S|P:X}` Substitutes `X` if either `-S` or `-P` was given to GCC. This may be combined with '`!`' and '`.`' sequences as well, although they have a stronger binding than the '`|`'. For example a spec string like this:

```
%{.c:-foo} %{!.c:-bar} %{.c|d:-baz} %{!.c|d:-boggle}
```

will output the following command-line options from the following input command-line options:

```
fred.c        -foo -baz
jim.d         -bar -boggle
-d fred.c     -foo -baz -boggle
-d jim.d      -bar -baz -boggle
```

The conditional text `X` in a `%{S:X}` or `%{!S:X}` construct may contain other nested '`%`' constructs or spaces, or even newlines. They are processed as usual, as described above.

The '`-O`', '`-f`', '`-m`', and '`-W`' switches are handled specifically in these constructs. If another value of '`-O`' or the negated form of a '`-f`', '`-m`', or '`-W`' switch is found later in the command line, the earlier switch value is ignored, except with `{S*}` where `S` is just one letter, which passes all matching options.

The character '`|`' at the beginning of the predicate text is used to indicate that a command should be piped to the following command, but only if '`-pipe`' is specified.

It is built into GCC which switches take arguments and which do not. (You might think it would be useful to generalize this to allow each compiler's spec to say which switches take arguments. But this cannot be done in a consistent fashion. GCC cannot even decide which input files have been specified without knowing which switches take arguments, and it must know which input files to compile in order to tell which compilers to run).

GCC also knows implicitly that arguments starting in '`-l`' are to be treated as compiler output files, and passed to the linker in their proper position among the other output files.

3.16 Specifying Target Machine and Compiler Version

By default, GCC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GCC, for different target machines, can be installed side by side. Then you specify which one to use with the '-b' option.

In addition, older and newer versions of GCC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

`-b` *machine*
: The argument *machine* specifies the target machine for compilation. This is useful when you have installed GCC as a cross-compiler.

  The value to use for *machine* is the same as was specified as the machine type when configuring GCC as a cross-compiler. For example, if a cross-compiler was configured with '`configure i386v`', meaning to compile for an 80386 running System V, then you would specify '`-b i386v`' to run that cross compiler.

  When you do not specify '`-b`', it normally means to compile for the same type of machine that you are using.

`-V` *version*
: The argument *version* specifies which version of GCC to run. This is useful when multiple versions are installed. For example, *version* might be '`2.0`', meaning to run GCC version 2.0.

  The default version, when you do not specify '`-V`', is the last version of GCC that you installed.

The '`-b`' and '`-V`' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GCC, for a given target machine, is normally kept in the directory '`/usr/local/lib/gcc-lib/`*machine*`/`*version*'.

Thus, sites can customize the effect of '`-b`' or '`-V`' either by changing the names of these directories or adding alternate names (or symbolic links). If in directory '`/usr/local/lib/gcc-lib/`' the file '`80386`' is a link to the file '`i386v`', then '`-b 80386`' becomes an alias for '`-b i386v`'.

In one respect, the '`-b`' or '`-V`' do not completely change to a different compiler: the top-level driver program `gcc` that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target. It is common for the interface to the other executables to change incompatibly between compiler versions, so unless the version specified is very close to that of the driver (for example, '`-V 3.0`' with a driver program from GCC version 3.0.1), use of '`-V`' may not work; for example, using '`-V 2.95.2`' will not work with a driver program from GCC 3.0.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine, and sufficiently similar compiler versions.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as `ogcc` and that for version 2.1 is installed as `gcc`, then the command `gcc` will use version 2.1 by default, while `ogcc` will use 2.0 by default. However, you can choose either version with either command with the '`-V`' option.

3.17 Hardware Models and Configurations

Earlier we discussed the standard option '`-b`' which chooses among different installed compilers for completely different target machines, such as VAX vs. 68000 vs. 80386.

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

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

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

3.17.1 M680x0 Options

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

`-m68000`
`-mc68000`
: Generate output for a 68000. This is the default when the compiler is configured for 68000-based systems.

  Use this option for microcontrollers with a 68000 or EC000 core, including the 68008, 68302, 68306, 68307, 68322, 68328 and 68356.

`-m68020`
`-mc68020`
: Generate output for a 68020. This is the default when the compiler is configured for 68020-based systems.

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

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

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

  This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. Use this option if your 68040 does not have code to emulate those instructions.

`-m68060` Generate output for a 68060. This is the default when the compiler is configured for 68060-based systems.

This option inhibits the use of 68020 and 68881/68882 instructions that have to be emulated by software on the 68060. Use this option if your 68060 does not have code to emulate those instructions.

`-mcpu32` Generate output for a CPU32. This is the default when the compiler is configured for CPU32-based systems.

Use this option for microcontrollers with a CPU32 or CPU32+ core, including the 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349 and 68360.

`-m5200` Generate output for a 520X "coldfire" family cpu. This is the default when the compiler is configured for 520X-based systems.

Use this option for microcontroller with a 5200 core, including the MCF5202, MCF5203, MCF5204 and MCF5202.

`-m68020-40`

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

`-m68020-60`

Generate output for a 68060, without using any of the new instructions. This results in code which can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68060.

`-mfpa` Generate output containing Sun FPA instructions for floating point.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all m68k targets. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets '`m68k-*-aout`' and '`m68k-*-coff`' do provide software floating point support.

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

`-mnobitfield`

Do not use the bit-field instructions. The '`-m68000`', '`-mcpu32`' and '`-m5200`' options imply '`-mnobitfield`'.

`-mbitfield`

Do use the bit-field instructions. The '`-m68020`' option implies '`-mbitfield`'. This is the default if you use a configuration designed for a 68020.

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

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

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

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

The `rtd` instruction is supported by the 68010, 68020, 68030, 68040, 68060 and CPU32 processors, but not by the 68000 or 5200.

`-malign-int`
`-mno-align-int`
: Control whether GCC aligns `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 32-bit boundary ('`-malign-int`') or a 16-bit boundary ('`-mno-align-int`'). Aligning variables on 32-bit boundaries produces code that runs somewhat faster on processors with 32-bit busses at the expense of more memory.

Warning: if you use the '`-malign-int`' switch, GCC will align structures containing the above types differently than most published application binary interface specifications for the m68k.

`-mpcrel`
: Use the pc-relative addressing mode of the 68000 directly, instead of using a global offset table. At present, this option implies '`-fpic`', allowing at most a 16-bit offset for pc-relative addressing. '`-fPIC`' is not presently supported with '`-mpcrel`', though this could be supported for 68020 and higher processors.

`-mno-strict-align`
`-mstrict-align`
: Do not (do) assume that unaligned memory references will be handled by the system.

3.17.2 M68hc1x Options

These are the '`-m`' options defined for the 68hc11 and 68hc12 microcontrollers. The default values for these options depends on which style of microcontroller was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m6811`
`-m68hc11`
: Generate output for a 68HC11. This is the default when the compiler is configured for 68HC11-based systems.

`-m6812`
`-m68hc12`
: Generate output for a 68HC12. This is the default when the compiler is configured for 68HC12-based systems.

`-mauto-incdec`
: Enable the use of 68HC12 pre and post auto-increment and auto-decrement addressing modes.

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

`-msoft-reg-count=`*count*
: Specify the number of pseudo-soft registers which are used for the code generation. The maximum number is 32. Using more pseudo-soft register may or may not result in better code depending on the program. The default is 4 for 68HC11 and 2 for 68HC12.

3.17.3 VAX Options

These '`-m`' options are defined for the VAX:

`-munix` Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the VAX cannot handle across long ranges.

`-mgnu` Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

`-mg` Output code for g-format floating point numbers instead of d-format.

3.17.4 SPARC Options

These '`-m`' switches are supported on the SPARC:

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

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

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

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

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

`-mhard-quad-float`
: Generate output containing quad-word (long double) floating point instructions.

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

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

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

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

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

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

`-mno-faster-structs`
`-mfaster-structs`
: With '`-mfaster-structs`', the compiler assumes that structures should have 8 byte alignment. This enables the use of pairs of `ldd` and `std` instructions for copies in structure assignment, in place of twice as many `ld` and `st` pairs. However, the use of this changed alignment directly violates the Sparc ABI. Thus, it's intended only for use on targets where the developer acknowledges that their resulting code will not be directly in line with the rules of the ABI.

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

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

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

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

These options are deprecated and will be deleted in a future GCC release. They have been replaced with '`-mcpu=xxx`'.

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

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

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

These options are deprecated and will be deleted in a future GCC release. They have been replaced with '`-mcpu=xxx`'.

`-mcpu=`*cpu_type*
Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`v7`', '`cypress`', '`v8`', '`supersparc`', '`sparclite`', '`hypersparc`', '`sparclite86x`', '`f930`', '`f934`', '`sparclet`', '`tsc701`', '`v9`', and '`ultrasparc`'.

Default instruction scheduling parameters are used for values that select an architecture and not an implementation. These are '`v7`', '`v8`', '`sparclite`', '`sparclet`', '`v9`'.

Here is a list of each supported architecture and their supported implementations.

```
v7:             cypress
v8:             supersparc, hypersparc
sparclite:      f930, f934, sparclite86x
sparclet:       tsc701
v9:             ultrasparc
```

`-mtune=`*cpu_type*
Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the instruction set or register set that the option '`-mcpu=`*cpu_type*' would.

The same values for '`-mcpu=`*cpu_type*' can be used for '`-mtune=`*cpu_type*', but the only useful values are those that select a particular cpu implementation. Those are '`cypress`', '`supersparc`', '`hypersparc`', '`f930`', '`f934`', '`sparclite86x`', '`tsc701`', and '`ultrasparc`'.

These '`-m`' switches are supported in addition to the above on the SPARCLET processor.

`-mlittle-endian`
Generate code for a processor running in little-endian mode.

`-mlive-g0`
Treat register `%g0` as a normal register. GCC will continue to clobber it as necessary but will not assume it always reads as 0.

`-mbroken-saverestore`
: Generate code that does not use non-trivial forms of the `save` and `restore` instructions. Early versions of the SPARCLET processor do not correctly handle `save` and `restore` instructions used with arguments. They correctly handle them used without arguments. A `save` instruction used without arguments increments the current window pointer but does not allocate a new stack frame. It is assumed that the window overflow trap handler will properly handle this case as will interrupt handlers.

These '`-m`' switches are supported in addition to the above on SPARC V9 processors in 64-bit environments.

`-mlittle-endian`
: Generate code for a processor running in little-endian mode.

`-m32`
`-m64`
: Generate code for a 32-bit or 64-bit environment. The 32-bit environment sets int, long and pointer to 32 bits. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits.

`-mcmodel=medlow`
: Generate code for the Medium/Low code model: the program must be linked in the low 32 bits of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked.

`-mcmodel=medmid`
: Generate code for the Medium/Middle code model: the program must be linked in the low 44 bits of the address space, the text segment must be less than 2G bytes, and data segment must be within 2G of the text segment. Pointers are 64 bits.

`-mcmodel=medany`
: Generate code for the Medium/Anywhere code model: the program may be linked anywhere in the address space, the text segment must be less than 2G bytes, and data segment must be within 2G of the text segment. Pointers are 64 bits.

`-mcmodel=embmedany`
: Generate code for the Medium/Anywhere code model for embedded systems: assume a 32-bit text and a 32-bit data segment, both starting anywhere (determined at link time). Register %g4 points to the base of the data segment. Pointers are still 64 bits. Programs are statically linked, PIC is not supported.

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

3.17.5 Convex Options

These '`-m`' options are defined for Convex:

`-mc1` Generate output for C1. The code will run on any Convex machine. The preprocessor symbol `__convex__c1__` is defined.

`-mc2` Generate output for C2. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C2. The preprocessor symbol `__convex_c2__` is defined.

`-mc32` Generate output for C32xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C32. The preprocessor symbol `__convex_c32__` is defined.

`-mc34` Generate output for C34xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C34. The preprocessor symbol `__convex_c34__` is defined.

`-mc38` Generate output for C38xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C38. The preprocessor symbol `__convex_c38__` is defined.

`-margcount`
Generate code which puts an argument count in the word preceding each argument list. This is compatible with regular CC, and a few programs may need the argument count word. GDB and other source-level debuggers do not need it; this info is in the symbol table.

`-mnoargcount`
Omit the argument count word. This is the default.

`-mvolatile-cache`
Allow volatile references to be cached. This is the default.

`-mvolatile-nocache`
Volatile references bypass the data cache, going all the way to memory. This is only needed for multi-processor code that does not use standard synchronization instructions. Making non-volatile references to volatile locations will not necessarily work.

`-mlong32` Type long is 32 bits, the same as type int. This is the default.

`-mlong64` Type long is 64 bits, the same as type long long. This option is useless, because no library support exists for it.

3.17.6 AMD29K Options

These '`-m`' options are defined for the AMD Am29000:

`-mdw` Generate code that assumes the `DW` bit is set, i.e., that byte and halfword operations are directly supported by the hardware. This is the default.

`-mndw` Generate code that assumes the `DW` bit is not set.

`-mbw` Generate code that assumes the system supports byte and halfword write operations. This is the default.

`-mnbw` Generate code that assumes the systems does not support byte and halfword write operations. '`-mnbw`' implies '`-mndw`'.

`-msmall` Use a small memory model that assumes that all function addresses are either within a single 256 KB segment or at an absolute address of less than 256k. This allows the `call` instruction to be used instead of a `const`, `consth`, `calli` sequence.

`-mnormal` Use the normal memory model: Generate `call` instructions only when calling functions in the same file and `calli` instructions otherwise. This works if each file occupies less than 256 KB but allows the entire executable to be larger than 256 KB. This is the default.

`-mlarge` Always use `calli` instructions. Specify this option if you expect a single file to compile into more than 256 KB of code.

`-m29050` Generate code for the Am29050.

`-m29000` Generate code for the Am29000. This is the default.

`-mkernel-registers`
: Generate references to registers `gr64-gr95` instead of to registers `gr96-gr127`. This option can be used when compiling kernel code that wants a set of global registers disjoint from that used by user-mode code.

  Note that when this option is used, register names in '`-f`' flags must use the normal, user-mode, names.

`-muser-registers`
: Use the normal set of global registers, `gr96-gr127`. This is the default.

`-mstack-check`
`-mno-stack-check`
: Insert (or do not insert) a call to `__msp_check` after each stack adjustment. This is often used for kernel code.

`-mstorem-bug`
`-mno-storem-bug`
: '`-mstorem-bug`' handles 29k processors which cannot handle the separation of a mtsrim insn and a storem instruction (most 29000 chips to date, but not the 29050).

`-mno-reuse-arg-regs`
`-mreuse-arg-regs`
: '`-mno-reuse-arg-regs`' tells the compiler to only use incoming argument registers for copying out arguments. This helps detect calling a function with fewer arguments than it was declared with.

`-mno-impure-text`
`-mimpure-text`
: '`-mimpure-text`', used in addition to '`-shared`', tells the compiler to not pass '`-assert pure-text`' to the linker when linking a shared object.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation.

You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mno-multm`
: Do not generate multm or multmu instructions. This is useful for some embedded systems which do not have trap handlers for these instructions.

3.17.7 ARM Options

These '`-m`' options are defined for Advanced RISC Machines (ARM) architectures:

`-mapcs-frame`
: Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying '`-fomit-frame-pointer`' with this option will cause the stack frames not to be generated for leaf functions. The default is '`-mno-apcs-frame`'.

`-mapcs`
: This is a synonym for '`-mapcs-frame`'.

`-mapcs-26`
: Generate code for a processor running with a 26-bit program counter, and conforming to the function calling standards for the APCS 26-bit option. This option replaces the '`-m2`' and '`-m3`' options of previous releases of the compiler.

`-mapcs-32`
: Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option. This option replaces the '`-m6`' option of previous releases of the compiler.

`-mthumb-interwork`
: Generate code which supports calling between the ARM and Thumb instruction sets. Without this option the two instruction sets cannot be reliably used inside one program. The default is '`-mno-thumb-interwork`', since slightly larger code is generated when '`-mthumb-interwork`' is specified.

`-mno-sched-prolog`
: Prevent the reordering of instructions in the function prolog, or the merging of those instruction with the instructions in the function's body. This means that all functions will start with a recognizable set of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is '`-msched-prolog`'.

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

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all ARM targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

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

`-mlittle-endian`
: Generate code for a processor running in little-endian mode. This is the default for all standard configurations.

`-mbig-endian`
: Generate code for a processor running in big-endian mode; the default is to compile code for a little-endian processor.

`-mwords-little-endian`
: This option only applies when generating code for big-endian processors. Generate code for a little-endian word order but a big-endian byte order. That is, a byte order of the form '`32107654`'. Note: this option should only be used if you require compatibility with code for big-endian ARM processors generated by versions of the compiler prior to 2.8.

`-malignment-traps`
: Generate code that will not trap if the MMU has alignment traps enabled. On ARM architectures prior to ARMv4, there were no instructions to access half-word objects stored in memory. However, when reading from memory a feature of the ARM architecture allows a word load to be used, even if the address is unaligned, and the processor core will rotate the data as it is being loaded. This option tells the compiler that such misaligned accesses will cause a MMU trap and that it should instead synthesise the access as a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary.

  This option is ignored when compiling for ARM architecture 4 or later, since these processors have instructions to directly access half-word objects in memory.

`-mno-alignment-traps`
: Generate code that assumes that the MMU will not trap unaligned accesses. This produces better code when the target instruction set does not have half-word memory operations (i.e. implementations prior to ARMv4).

  Note that you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory.

  The default setting for most targets is '`-mno-alignment-traps`', since this produces better code when there are no half-word memory instructions available.

`-mshort-load-bytes`
`-mno-short-load-words`
: These are deprecated aliases for '`-malignment-traps`'.

`-mno-short-load-bytes`
`-mshort-load-words`
: This are deprecated aliases for '`-mno-alignment-traps`'.

`-mbsd` This option only applies to RISC iX. Emulate the native BSD-mode compiler. This is the default if '`-ansi`' is not specified.

`-mxopen` This option only applies to RISC iX. Emulate the native X/Open-mode compiler.

`-mno-symrename`
: This option only applies to RISC iX. Do not run the assembler post-processor, '`symrename`', after code has been assembled. Normally it is necessary to modify some of the standard symbols in preparation for linking with the RISC iX C library; this option suppresses this pass. The post-processor is never run when the compiler is built for cross-compilation.

`-mcpu=`*name*
: This specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. Permissible names are: '`arm2`', '`arm250`', '`arm3`', '`arm6`', '`arm60`', '`arm600`', '`arm610`', '`arm620`', '`arm7`', '`arm7m`', '`arm7d`', '`arm7dm`', '`arm7di`', '`arm7dmi`', '`arm70`', '`arm700`', '`arm700i`', '`arm710`', '`arm710c`', '`arm7100`', '`arm7500`', '`arm7500fe`', '`arm7tdmi`', '`arm8`', '`strongarm`', '`strongarm110`', '`strongarm1100`', '`arm8`', '`arm810`', '`arm9`', '`arm9e`', '`arm920`', '`arm920t`', '`arm940t`', '`arm9tdmi`', '`arm10tdmi`', '`arm1020t`', '`xscale`'.

`-mtune=`*name*
: This option is very similar to the '`-mcpu=`' option, except that instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the cpu specified by a '`-mcpu=`' option. For some ARM implementations better performance can be obtained by using this option.

`-march=`*name*
: This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. This option can be used in conjunction with or instead of the '`-mcpu=`' option. Permissible names are: '`armv2`', '`armv2a`', '`armv3`', '`armv3m`', '`armv4`', '`armv4t`', '`armv5`', '`armv5t`', '`armv5te`'.

`-mfpe=`*number*
`-mfp=`*number*
: This specifies the version of the floating point emulation available on the target. Permissible values are 2 and 3. '`-mfp=`' is a synonym for '`-mfpe=`', for compatibility with older versions of GCC.

`-mstructure-size-boundary=`*n*
: The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissible values are 8 and 32. The default value varies for different toolchains. For the COFF targeted toolchain the default value is 8. Specifying the larger number can produce faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect

to work with code or libraries compiled with the other value, if they exchange information using structures or unions.

`-mabort-on-noreturn`
: Generate a call to the function `abort` at the end of a `noreturn` function. It will be executed if the function tries to return.

`-mlong-calls`
`-mno-long-calls`
: Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction.

Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the '`short-call`' attribute, functions that are inside the scope of a '`#pragma no_long_calls`' directive and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the '`long-call`' attribute or the '`section`' attribute, and functions that are within the scope of a '`#pragma long_calls`' directive, will always be turned into long calls.

This feature is not enabled by default. Specifying '`-mno-long-calls`' will restore the default behavior, as will placing the function calls within the scope of a '`#pragma long_calls_off`' directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.

`-mnop-fun-dllimport`
: Disable support for the `dllimport` attribute.

`-msingle-pic-base`
: Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The run-time system is responsible for initializing this register with an appropriate value before execution begins.

`-mpic-register=`*reg*
: Specify the register to be used for PIC addressing. The default is R10 unless stack-checking is enabled, when R9 is used.

`-mpoke-function-name`
: Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this:

```
t0
    .ascii "arm_poke_function_name", 0
    .align
t1
    .word 0xff000000 + (t1 - t0)
arm_poke_function_name
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
```

When performing a stack backtrace, code can inspect the value of `pc` stored at `fp + 0`. If the trace function then looks at location `pc - 12` and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length `((pc[-3]) & 0xff000000)`.

`-mthumb`
: Generate code for the 16-bit Thumb instruction set. The default is to use the 32-bit ARM instruction set.

`-mtpcs-frame`
: Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all non-leaf functions. (A leaf function is one that does not call any other functions.) The default is '`-mno-tpcs-frame`'.

`-mtpcs-leaf-frame`
: Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all leaf functions. (A leaf function is one that does not call any other functions.) The default is '`-mno-apcs-leaf-frame`'.

`-mcallee-super-interworking`
: Gives all externally visible functions in the file being compiled an ARM instruction set header which switches to Thumb mode before executing the rest of the function. This allows these functions to be called from non-interworking code.

`-mcaller-super-interworking`
: Allows calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. There is a small overhead in the cost of executing a function pointer if this option is enabled.

3.17.8 MN10200 Options

These '`-m`' options are defined for Matsushita MN10200 architectures:

`-mrelax`
: Indicate to the linker that it should perform a relaxation optimization pass to shorten branches, calls and absolute memory addresses. This option only has an effect when used on the command line for the final link step.

  This option makes symbolic debugging impossible.

3.17.9 MN10300 Options

These '`-m`' options are defined for Matsushita MN10300 architectures:

`-mmult-bug`
: Generate code to avoid bugs in the multiply instructions for the MN10300 processors. This is the default.

`-mno-mult-bug`
: Do not generate code to avoid bugs in the multiply instructions for the MN10300 processors.

`-mam33`
: Generate code which uses features specific to the AM33 processor.

`-mno-am33`
: Do not generate code which uses features specific to the AM33 processor. This is the default.

`-mno-crt0`
: Do not link in the C run-time initialization object file.

`-mrelax`
: Indicate to the linker that it should perform a relaxation optimization pass to shorten branches, calls and absolute memory addresses. This option only has an effect when used on the command line for the final link step.

  This option makes symbolic debugging impossible.

3.17.10 M32R/D Options

These '`-m`' options are defined for Mitsubishi M32R/D architectures:

`-m32rx`
: Generate code for the M32R/X.

`-m32r`
: Generate code for the M32R. This is the default.

`-mcode-model=small`
: Assume all objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and assume all subroutines are reachable with the `bl` instruction. This is the default.

  The addressability of a particular object can be set with the `model` attribute.

`-mcode-model=medium`
: Assume objects may be anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and assume all subroutines are reachable with the `bl` instruction.

`-mcode-model=large`
: Assume objects may be anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and assume subroutines may not be reachable with the `bl` instruction (the compiler will generate the much slower `seth/add3/jl` instruction sequence).

`-msdata=none`
: Disable use of the small data area. Variables will be put into one of '`.data`', '`bss`', or '`.rodata`' (unless the `section` attribute has been specified). This is the default.

  The small data area consists of sections '`.sdata`' and '`.sbss`'. Objects may be explicitly put in the small data area with the `section` attribute using one of these sections.

`-msdata=sdata`
: Put small global and static data in the small data area, but do not generate special code to reference them.

`-msdata=use`
: Put small global and static data in the small data area, and generate special instructions to reference them.

`-G num` Put global and static objects less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss sections. The default value of *num* is 8. The '`-msdata`' option must be set to one of '`sdata`' or '`use`' for this option to have any effect.

All modules should be compiled with the same '`-G num`' value. Compiling with different values of *num* may or may not work; if it doesn't the linker will give an error message—incorrect code will not be generated.

3.17.11 M88K Options

These '`-m`' options are defined for Motorola 88k architectures:

`-m88000` Generate code that works well on both the m88100 and the m88110.

`-m88100` Generate code that works best for the m88100, but that also runs on the m88110.

`-m88110` Generate code that works best for the m88110, and may not run on the m88100.

`-mbig-pic`
Obsolete option to be removed from the next revision. Use '`-fPIC`'.

`-midentify-revision`
Include an `ident` directive in the assembler output recording the source file name, compiler name and version, timestamp, and compilation flags used.

`-mno-underscores`
In assembler output, emit symbol names without adding an underscore character at the beginning of each name. The default is to use an underscore as prefix on each name.

`-mocs-debug-info`
`-mno-ocs-debug-info`
Include (or omit) additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard, "OCS". This extra information allows debugging of code that has had the frame pointer eliminated. The default for DG/UX, SVr4, and Delta 88 SVr3.2 is to include this information; other 88k configurations omit this information by default.

`-mocs-frame-position`
When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address, which is the stack pointer (register 31) on entry to the function. The DG/UX, SVr4, Delta88 SVr3.2, and BCS configurations use '`-mocs-frame-position`'; other 88k configurations have the default '`-mno-ocs-frame-position`'.

`-mno-ocs-frame-position`
When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the -g switch.

`-moptimize-arg-area`
: Save space by reorganizing the stack frame. This option generates code that does not agree with the 88open specifications, but uses less memory.

`-mno-optimize-arg-area`
: Do not reorganize the stack frame to save space. This is the default. The generated conforms to the specification, but uses more memory.

`-mshort-data-`*num*
: Generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). You control which data references are affected by specifying *num* with this option. For example, if you specify '`-mshort-data-512`', then the data references affected are those involving displacements of less than 512 bytes. '`-mshort-data-`*num*' is not effective for *num* greater than 64k.

`-mserialize-volatile`
`-mno-serialize-volatile`
: Do, or don't, generate code to guarantee sequential consistency of volatile memory references. By default, consistency is guaranteed.

The order of memory references made by the MC88110 processor does not always match the order of the instructions requesting those references. In particular, a load instruction may execute before a preceding store instruction. Such reordering violates sequential consistency of volatile memory references, when there are multiple processors. When consistency must be guaranteed, GCC generates special instructions, as needed, to force execution in the proper order.

The MC88100 processor does not reorder memory references and so always provides sequential consistency. However, by default, GCC generates the special instructions to guarantee consistency even when you use '`-m88100`', so that the code may be run on an MC88110 processor. If you intend to run your code only on the MC88100 processor, you may use '`-mno-serialize-volatile`'.

The extra code generated to guarantee consistency may affect the performance of your application. If you know that you can safely forgo this guarantee, you may use '`-mno-serialize-volatile`'.

`-msvr4`
`-msvr3`
: Turn on ('`-msvr4`') or off ('`-msvr3`') compiler extensions related to System V release 4 (SVr4). This controls the following:

1. Which variant of the assembler syntax to emit.
2. '`-msvr4`' makes the C preprocessor recognize '`#pragma weak`' that is used on System V release 4.
3. '`-msvr4`' makes GCC issue additional declaration directives used in SVr4.

'`-msvr4`' is the default for the m88k-motorola-sysv4 and m88k-dg-dgux m88k configurations. '`-msvr3`' is the default for all other m88k configurations.

`-mversion-03.00`
: This option is obsolete, and is ignored.

`-mno-check-zero-division`
`-mcheck-zero-division`

Do, or don't, generate code to guarantee that integer division by zero will be detected. By default, detection is guaranteed.

Some models of the MC88100 processor fail to trap upon integer division by zero under certain conditions. By default, when compiling code that might be run on such a processor, GCC generates code that explicitly checks for zero-valued divisors and traps with exception number 503 when one is detected. Use of '`-mno-check-zero-division`' suppresses such checking for code generated to run on an MC88100 processor.

GCC assumes that the MC88110 processor correctly detects all instances of integer division by zero. When '`-m88110`' is specified, no explicit checks for zero-valued divisors are generated, and both '`-mcheck-zero-division`' and '`-mno-check-zero-division`' are ignored.

`-muse-div-instruction`

Use the div instruction for signed integer division on the MC88100 processor. By default, the div instruction is not used.

On the MC88100 processor the signed integer division instruction div) traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. By default, when compiling code that might be run on an MC88100 processor, GCC emulates signed integer division using the unsigned integer division instruction divu), thereby avoiding the large penalty of a trap to the operating system. Such emulation has its own, smaller, execution cost in both time and space. To the extent that your code's important signed integer division operations are performed on two nonnegative operands, it may be desirable to use the div instruction directly.

On the MC88110 processor the div instruction (also known as the divs instruction) processes negative operands without trapping to the operating system. When '`-m88110`' is specified, '`-muse-div-instruction`' is ignored, and the div instruction is used for signed integer division.

Note that the result of dividing `INT_MIN` by -1 is undefined. In particular, the behavior of such a division with and without '`-muse-div-instruction`' may differ.

`-mtrap-large-shift`
`-mhandle-large-shift`

Include code to detect bit-shifts of more than 31 bits; respectively, trap such shifts or emit code to handle them properly. By default GCC makes no special provision for large bit shifts.

`-mwarn-passed-structs`

Warn when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of portability problems. By default, GCC issues no such warning.

3.17.12 IBM RS/6000 and PowerPC Options

These '`-m`' options are defined for the IBM RS/6000 and PowerPC:

`-mpower`
`-mno-power`
`-mpower2`
`-mno-power2`
`-mpowerpc`
`-mno-powerpc`
`-mpowerpc-gpopt`
`-mno-powerpc-gpopt`
`-mpowerpc-gfxopt`
`-mno-powerpc-gfxopt`
`-mpowerpc64`
`-mno-powerpc64`

GCC supports two related instruction set architectures for the RS/6000 and PowerPC. The *POWER* instruction set are those instructions supported by the '`rios`' chip set used in the original RS/6000 systems and the *PowerPC* instruction set is the architecture of the Motorola MPC5xx, MPC6xx, MPC8xx microprocessors, and the IBM 4xx microprocessors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GCC. Specifying the '`-mcpu=`*cpu_type*' overrides the specification of these options. We recommend you use the '`-mcpu=`*cpu_type*' option rather than the options listed above.

The '`-mpower`' option allows GCC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying '`-mpower2`' implies '`-power`' and also allows GCC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The '`-mpowerpc`' option allows GCC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture. Specifying '`-mpowerpc-gpopt`' implies '`-mpowerpc`' and also allows GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying '`-mpowerpc-gfxopt`' implies '`-mpowerpc`' and also allows GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

The '`-mpowerpc64`' option allows GCC to generate the additional 64-bit instructions that are found in the full PowerPC64 architecture and to treat GPRs as 64-bit, doubleword quantities. GCC defaults to '`-mno-powerpc64`'.

If you specify both '`-mno-power`' and '`-mno-powerpc`', GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both

'-mpower' and '-mpowerpc' permits GCC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

-mnew-mnemonics
-mold-mnemonics
Select which mnemonics to use in the generated assembler code. With '-mnew-mnemonics', GCC uses the assembler mnemonics defined for the PowerPC architecture. With '-mold-mnemonics' it uses the assembler mnemonics defined for the POWER architecture. Instructions defined in only one architecture have only one mnemonic; GCC uses that mnemonic irrespective of which of these options is specified.

GCC defaults to the mnemonics appropriate for the architecture in use. Specifying '-mcpu=*cpu_type*' sometimes overrides the value of these option. Unless you are building a cross-compiler, you should normally not specify either '-mnew-mnemonics' or '-mold-mnemonics', but should instead accept the default.

-mcpu=*cpu_type*
Set architecture type, register usage, choice of mnemonics, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are 'rios', 'rios1', 'rsc', 'rios2', 'rs64a', '601', '602', '603', '603e', '604', '604e', '620', '630', '740', '7400', '7450', '750', 'power', 'power2', 'powerpc', '403', '505', '801', '821', '823', and '860' and 'common'.

'-mcpu=common' selects a completely generic processor. Code generated under this option will run on any POWER or PowerPC processor. GCC will use only the instructions in the common subset of both architectures, and will not use the MQ register. GCC assumes a generic processor model for scheduling purposes.

'-mcpu=power', '-mcpu=power2', '-mcpu=powerpc', and '-mcpu=powerpc64' specify generic POWER, POWER2, pure 32-bit PowerPC (i.e., not MPC601), and 64-bit PowerPC architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

The other options specify a specific processor. Code generated under those options will run best on that processor, and may not run at all on others.

The '-mcpu' options automatically enable or disable other '-m' options as follows:

'common' '-mno-power', '-mno-powerc'

'power'
'power2'
'rios1'
'rios2'
'rsc' '-mpower', '-mno-powerpc', '-mno-new-mnemonics'

'powerpc'
'rs64a'
'602'
'603'
'603e'
'604'
'620'
'630'
'740'
'7400'
'7450'
'750'
'505' '-mno-power', '-mpowerpc', '-mnew-mnemonics'

'601' '-mpower', '-mpowerpc', '-mnew-mnemonics'

'403'
'821'
'860' '-mno-power', '-mpowerpc', '-mnew-mnemonics', '-msoft-float'

`-mtune=`*cpu_type*
Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the architecture type, register usage, or choice of mnemonics, as '-mcpu=*cpu_type*' would. The same values for *cpu_type* are used for '-mtune' as for '-mcpu'. If both are specified, the code generated will use the architecture, registers, and mnemonics set by '-mcpu', but the scheduling parameters set by '-mtune'.

`-maltivec`
`-mno-altivec`
These switches enable or disable the use of built-in functions that allow access to the AltiVec instruction set. You may also need to set '-mabi=altivec' to adjust the current ABI with AltiVec ABI enhancements.

`-mfull-toc`
`-mno-fp-in-toc`
`-mno-sum-in-toc`
`-mminimal-toc`
Modify generation of the TOC (Table Of Contents), which is created for every executable file. The '-mfull-toc' option is selected by default. In that case, GCC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GCC will also place floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the '-mno-fp-in-toc' and '-mno-sum-in-toc' options. '-mno-fp-in-toc' prevents GCC from putting floating-point constants in the TOC and '-mno-sum-in-toc' forces GCC to generate code to calculate the sum of an address and a constant at run-time instead of putting that sum into the TOC. You may specify one or both of these options. Each causes GCC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify '-mminimal-toc' instead. This option causes GCC to make only one TOC entry for every file. When you specify this option, GCC will produce code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently executed code.

-maix64
-maix32
Enable 64-bit AIX ABI and calling convention: 64-bit pointers, 64-bit long type, and the infrastructure needed to support them. Specifying '-maix64' implies '-mpowerpc64' and '-mpowerpc', while '-maix32' disables the 64-bit ABI and implies '-mno-powerpc64'. GCC defaults to '-maix32'.

-mxl-call
-mno-xl-call
On AIX, pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument FPRs. The AIX calling convention was extended but not initially documented to handle an obscure K&R C case of calling a function that takes the address of its arguments with fewer arguments than declared. AIX XL compilers access floating point arguments which do not fit in the RSA from the stack when a subroutine is compiled without optimization. Because always storing floating-point arguments on the stack is inefficient and rarely needed, this option is not enabled by default and only is necessary when calling subroutines compiled by AIX XL compilers without optimization.

-mpe
Support *IBM RS/6000 SP Parallel Environment* (PE). Link an application written to use message passing with special startup code to enable the application to run. The system must have PE installed in the standard location ('/usr/lpp/ppe.poe/'), or the 'specs' file must be overridden with the '-specs=' option to specify the appropriate directory location. The Parallel Environment does not support threads, so the '-mpe' option and the '-pthread' option are incompatible.

-msoft-float
-mhard-float
Generate code that does not use (uses) the floating-point register set. Software floating point emulation is provided if you use the '-msoft-float' option, and pass the option to GCC when linking.

-mmultiple
-mno-multiple
Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '-mmultiple' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750 which permit the instructions usage in little endian mode.

`-mstring`
`-mno-string`
: Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '`-mstring`' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750 which permit the instructions usage in little endian mode.

`-mupdate`
`-mno-update`
: Generate code that uses (does not use) the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default. If you use '`-mno-update`', there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.

`-mfused-madd`
`-mno-fused-madd`
: Generate code that uses (does not use) the floating point multiply and accumulate instructions. These instructions are generated by default if hardware floating is used.

`-mno-bit-align`
`-mbit-align`
: On System V.4 and embedded PowerPC systems do not (do) force structures and unions that contain bit-fields to be aligned to the base type of the bit-field.

  For example, by default a structure containing nothing but 8 `unsigned` bit-fields of length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using '`-mno-bit-align`', the structure would be aligned to a 1 byte boundary and be one byte in size.

`-mno-strict-align`
`-mstrict-align`
: On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.

`-mrelocatable`
`-mno-relocatable`
: On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. If you use '`-mrelocatable`' on any module, all objects linked together must be compiled with '`-mrelocatable`' or '`-mrelocatable-lib`'.

`-mrelocatable-lib`
`-mno-relocatable-lib`
: On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. Modules compiled

with '-mrelocatable-lib' can be linked with either modules compiled without '-mrelocatable' and '-mrelocatable-lib' or with modules compiled with the '-mrelocatable' options.

-mno-toc
-mtoc
: On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

-mlittle
-mlittle-endian
: On System V.4 and embedded PowerPC systems compile code for the processor in little endian mode. The '-mlittle-endian' option is the same as '-mlittle'.

-mbig
-mbig-endian
: On System V.4 and embedded PowerPC systems compile code for the processor in big endian mode. The '-mbig-endian' option is the same as '-mbig'.

-mcall-sysv
: On System V.4 and embedded PowerPC systems compile code using calling conventions that adheres to the March 1995 draft of the System V Application Binary Interface, PowerPC processor supplement. This is the default unless you configured GCC using 'powerpc-*-eabiaix'.

-mcall-sysv-eabi
: Specify both '-mcall-sysv' and '-meabi' options.

-mcall-sysv-noeabi
: Specify both '-mcall-sysv' and '-mno-eabi' options.

-mcall-aix
: On System V.4 and embedded PowerPC systems compile code using calling conventions that are similar to those used on AIX. This is the default if you configured GCC using 'powerpc-*-eabiaix'.

-mcall-solaris
: On System V.4 and embedded PowerPC systems compile code for the Solaris operating system.

-mcall-linux
: On System V.4 and embedded PowerPC systems compile code for the Linux-based GNU system.

-mcall-gnu
: On System V.4 and embedded PowerPC systems compile code for the Hurd-based GNU system.

-mcall-netbsd
: On System V.4 and embedded PowerPC systems compile code for the NetBSD operating system.

-maix-struct-return
: Return all structures in memory (as specified by the AIX ABI).

`-msvr4-struct-return`
: Return structures smaller than 8 bytes in registers (as specified by the SVR4 ABI).

`-mabi=altivec`
: Extend the current ABI with AltiVec ABI extensions. This does not change the default ABI, instead it adds the AltiVec ABI extensions to the current ABI.

`-mabi=no-altivec`
: Disable AltiVec ABI extensions for the current ABI.

`-mprototype`
`-mno-prototype`
: On System V.4 and embedded PowerPC systems assume that all calls to variable argument functions are properly prototyped. Otherwise, the compiler must insert an instruction before every non prototyped call to set or clear bit 6 of the condition code register (*CR*) to indicate whether floating point values were passed in the floating point registers in case the function takes a variable arguments. With '`-mprototype`', only calls to prototyped variable argument functions will set or clear the bit.

`-msim`
: On embedded PowerPC systems, assume that the startup module is called '`sim-crt0.o`' and that the standard C libraries are '`libsim.a`' and '`libc.a`'. This is the default for '`powerpc-*-eabisim`'. configurations.

`-mmvme`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libmvme.a`' and '`libc.a`'.

`-mads`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libads.a`' and '`libc.a`'.

`-myellowknife`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libyk.a`' and '`libc.a`'.

`-mvxworks`
: On System V.4 and embedded PowerPC systems, specify that you are compiling for a VxWorks system.

`-memb`
: On embedded PowerPC systems, set the *PPC_EMB* bit in the ELF flags header to indicate that '`eabi`' extended relocations are used.

`-meabi`
`-mno-eabi`
: On System V.4 and embedded PowerPC systems do (do not) adhere to the Embedded Applications Binary Interface (eabi) which is a set of modifications to the System V.4 specifications. Selecting '`-meabi`' means that the stack is aligned to an 8 byte boundary, a function `__eabi` is called to from `main` to set up the eabi environment, and the '`-msdata`' option can use both `r2` and `r13` to point to two separate small data areas. Selecting '`-mno-eabi`' means that the stack is aligned to a 16 byte boundary, do not call an initialization function from `main`, and the '`-msdata`' option will only use `r13` to point to a single small

data area. The '`-meabi`' option is on by default if you configured GCC using one of the '`powerpc*-*-eabi*`' options.

`-msdata=eabi`
: On System V.4 and embedded PowerPC systems, put small initialized `const` global and static data in the '`.sdata2`' section, which is pointed to by register `r2`. Put small initialized non-`const` global and static data in the '`.sdata`' section, which is pointed to by register `r13`. Put small uninitialized global and static data in the '`.sbss`' section, which is adjacent to the '`.sdata`' section. The '`-msdata=eabi`' option is incompatible with the '`-mrelocatable`' option. The '`-msdata=eabi`' option also sets the '`-memb`' option.

`-msdata=sysv`
: On System V.4 and embedded PowerPC systems, put small global and static data in the '`.sdata`' section, which is pointed to by register `r13`. Put small uninitialized global and static data in the '`.sbss`' section, which is adjacent to the '`.sdata`' section. The '`-msdata=sysv`' option is incompatible with the '`-mrelocatable`' option.

`-msdata=default`
`-msdata`
: On System V.4 and embedded PowerPC systems, if '`-meabi`' is used, compile code the same as '`-msdata=eabi`', otherwise compile code the same as '`-msdata=sysv`'.

`-msdata-data`
: On System V.4 and embedded PowerPC systems, put small global and static data in the '`.sdata`' section. Put small uninitialized global and static data in the '`.sbss`' section. Do not use register `r13` to address small data however. This is the default behavior unless other '`-msdata`' options are used.

`-msdata=none`
`-mno-sdata`
: On embedded PowerPC systems, put all initialized global and static data in the '`.data`' section, and all uninitialized data in the '`.bss`' section.

`-G` *num*
: On embedded PowerPC systems, put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. By default, *num* is 8. The '`-G` *num*' switch is also passed to the linker. All modules should be compiled with the same '`-G` *num*' value.

`-mregnames`
`-mno-regnames`
: On System V.4 and embedded PowerPC systems do (do not) emit register names in the assembly language output using symbolic forms.

`-pthread`
: Adds support for multithreading with the *pthreads* library. This option sets flags for both the preprocessor and linker.

3.17.13 IBM RT Options

These '`-m`' options are defined for the IBM RT PC:

`-min-line-mul`
: Use an in-line code sequence for integer multiplies. This is the default.

`-mcall-lib-mul`
: Call `lmul$$` for integer multiples.

`-mfull-fp-blocks`
: Generate full-size floating point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mminimum-fp-blocks`
: Do not include extra scratch space in floating point data blocks. This results in smaller code, but slower execution, since scratch space must be allocated dynamically.

`-mfp-arg-in-fpregs`
: Use a calling sequence incompatible with the IBM calling convention in which floating point arguments are passed in floating point registers. Note that `varargs.h` and `stdarg.h` will not work with floating point operands if this option is specified.

`-mfp-arg-in-gregs`
: Use the normal calling convention for floating point arguments. This is the default.

`-mhc-struct-return`
: Return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. Use the option '`-fpcc-struct-return`' for compatibility with the Portable C Compiler (pcc).

`-mnohc-struct-return`
: Return some structures of more than one word in registers, when convenient. This is the default. For compatibility with the IBM-supplied compilers, use the option '`-fpcc-struct-return`' or the option '`-mhc-struct-return`'.

3.17.14 MIPS Options

These '`-m`' options are defined for the MIPS family of computers:

`-march=`*cpu-type*
: Assume the defaults for the machine type *cpu-type* when generating instructions. The choices for *cpu-type* are '`r2000`', '`r3000`', '`r3900`', '`r4000`', '`r4100`', '`r4300`', '`r4400`', '`r4600`', '`r4650`', '`r5000`', '`r6000`', '`r8000`', and '`orion`'. Additionally, the '`r2000`', '`r3000`', '`r4000`', '`r5000`', and '`r6000`' can be abbreviated as '`r2k`' (or '`r2K`'), '`r3k`', etc.

`-mtune=`*cpu-type*
: Assume the defaults for the machine type *cpu-type* when scheduling instructions. The choices for *cpu-type* are '`r2000`', '`r3000`', '`r3900`', '`r4000`', '`r4100`', '`r4300`', '`r4400`', '`r4600`', '`r4650`', '`r5000`', '`r6000`', '`r8000`', and '`orion`'. Additionally, the '`r2000`', '`r3000`', '`r4000`', '`r5000`', and '`r6000`' can be abbreviated

as '`r2k`' (or '`r2K`'), '`r3k`', etc. While picking a specific *cpu-type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) without a '`-mipsX`' or '`-mabi`' switch being used.

`-mcpu=`*cpu-type*
: This is identical to specifying both '`-march`' and '`-mtune`'.

`-mips1`
: Issue instructions from level 1 of the MIPS ISA. This is the default. '`r3000`' is the default *cpu-type* at this ISA level.

`-mips2`
: Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). '`r6000`' is the default *cpu-type* at this ISA level.

`-mips3`
: Issue instructions from level 3 of the MIPS ISA (64-bit instructions). '`r4000`' is the default *cpu-type* at this ISA level.

`-mips4`
: Issue instructions from level 4 of the MIPS ISA (conditional move, prefetch, enhanced FPU instructions). '`r8000`' is the default *cpu-type* at this ISA level.

`-mfp32`
: Assume that 32 32-bit floating point registers are available. This is the default.

`-mfp64`
: Assume that 32 64-bit floating point registers are available. This is the default when the '`-mips3`' option is used.

`-mfused-madd`
`-mno-fused-madd`
: Generate code that uses (does not use) the floating point multiply and accumulate instructions, when they are available. These instructions are generated by default if they are available, but this may be undesirable if the extra precision causes problems or on certain chips in the mode where denormals are rounded to zero where denormals generated by multiply and accumulate instructions cause exceptions anyway.

`-mgp32`
: Assume that 32 32-bit general purpose registers are available. This is the default.

`-mgp64`
: Assume that 32 64-bit general purpose registers are available. This is the default when the '`-mips3`' option is used.

`-mint64`
: Force int and long types to be 64 bits wide. See '`-mlong32`' for an explanation of the default, and the width of pointers.

`-mlong64`
: Force long types to be 64 bits wide. See '`-mlong32`' for an explanation of the default, and the width of pointers.

`-mlong32`
: Force long, int, and pointer types to be 32 bits wide.

  If none of '`-mlong32`', '`-mlong64`', or '`-mint64`' are set, the size of ints, longs, and pointers depends on the ABI and ISA chosen. For '`-mabi=32`', and '`-mabi=n32`', ints and longs are 32 bits wide. For '`-mabi=64`', ints are 32 bits, and longs are 64 bits wide. For '`-mabi=eabi`' and either '`-mips1`' or '`-mips2`', ints and longs are 32 bits wide. For '`-mabi=eabi`' and higher ISAs, ints are 32 bits, and longs are 64 bits wide. The width of pointer types is the smaller of the width of longs or the width of general purpose registers (which in turn depends on the ISA).

`-mabi=32`
`-mabi=o64`
`-mabi=n32`
`-mabi=64`
`-mabi=eabi`
: Generate code for the indicated ABI. The default instruction level is '`-mips1`' for '`32`', '`-mips3`' for '`n32`', and '`-mips4`' otherwise. Conversely, with '`-mips1`' or '`-mips2`', the default ABI is '`32`'; otherwise, the default ABI is '`64`'.

`-mmips-as`
: Generate code for the MIPS assembler, and invoke '`mips-tfile`' to add normal debug information. This is the default for all platforms except for the OSF/1 reference platform, using the OSF/rose object format. If the either of the '`-gstabs`' or '`-gstabs+`' switches are used, the '`mips-tfile`' program will encapsulate the stabs within MIPS ECOFF.

`-mgas`
: Generate code for the GNU assembler. This is the default on the OSF/1 reference platform, using the OSF/rose object format. Also, this is the default if the configure option '`--with-gnu-as`' is used.

`-msplit-addresses`
`-mno-split-addresses`
: Generate code to load the high and low parts of address constants separately. This allows GCC to optimize away redundant loads of the high order bits of addresses. This optimization requires GNU as and GNU ld. This optimization is enabled by default for some embedded targets where GNU as and GNU ld are standard.

`-mrnames`
`-mno-rnames`
: The '`-mrnames`' switch says to output code using the MIPS software names for the registers, instead of the hardware names (ie, *a0* instead of *$4*). The only known assembler that supports this option is the Algorithmics assembler.

`-mgpopt`
`-mno-gpopt`
: The '`-mgpopt`' switch says to write all of the data declarations before the instructions in the text section, this allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

`-mstats`
`-mno-stats`
: For each non-inline function processed, the '`-mstats`' switch causes the compiler to emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`
`-mno-memcpy`
: The '`-mmemcpy`' switch makes all block moves call the appropriate string function ('`memcpy`' or '`bcopy`') instead of possibly generating inline code.

`-mmips-tfile`
`-mno-mips-tfile`
: The '`-mno-mips-tfile`' switch causes the compiler not postprocess the object file with the '`mips-tfile`' program, after the MIPS assembler has generated it to add debug support. If '`mips-tfile`' is not run, then no local variables will be available to the debugger. In addition, '`stage2`' and '`stage3`' objects will have the temporary file names passed to the assembler embedded in the object file, which means the objects will not compare the same. The '`-mno-mips-tfile`' switch should only be used when there are bugs in the '`mips-tfile`' program that prevents compilation.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mhard-float`
: Generate output containing floating point instructions. This is the default if you use the unmodified sources.

`-mabicalls`
`-mno-abicalls`
: Emit (or do not emit) the pseudo operations '`.abicalls`', '`.cpload`', and '`.cprestore`' that some System V.4 ports use for position independent code.

`-mlong-calls`
`-mno-long-calls`
: Do all calls with the '`JALR`' instruction, which requires loading up a function's address into a register before the call. You need to use this switch, if you call outside of the current 512 megabyte segment to functions that are not through pointers.

`-mhalf-pic`
`-mno-half-pic`
: Put pointers to extern references into the data section and load them up, rather than put the references in the text section.

`-membedded-pic`
`-mno-embedded-pic`
: Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the $gp register. No more than 65536 bytes of global data may be used. This requires GNU as and GNU ld which do most of the work. This currently only works on targets which use ECOFF; it does not work with ELF.

`-membedded-data`
`-mno-embedded-data`
: Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code

than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-muninit-const-in-rodata`
`-mno-uninit-const-in-rodata`
: When used together with '`-membedded-data`', it will always store uninitialized const variables in the read-only data section.

`-msingle-float`
`-mdouble-float`
: The '`-msingle-float`' switch tells gcc to assume that the floating point coprocessor only supports single precision operations, as on the '`r4650`' chip. The '`-mdouble-float`' switch permits gcc to use double precision operations. This is the default.

`-mmad`
`-mno-mad`
: Permit use of the '`mad`', '`madu`' and '`mul`' instructions, as on the '`r4650`' chip.

`-m4650`
: Turns on '`-msingle-float`', '`-mmad`', and, at least for now, '`-mcpu=r4650`'.

`-mips16`
`-mno-mips16`
: Enable 16-bit instructions.

`-mentry`
: Use the entry and exit pseudo ops. This option can only be used with '`-mips16`'.

`-EL`
: Compile code for the processor in little endian mode. The requisite libraries are assumed to exist.

`-EB`
: Compile code for the processor in big endian mode. The requisite libraries are assumed to exist.

`-G` *num*
: Put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (*gp* or *$28*), instead of the normal two words used. By default, *num* is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The '`-G` *num*' switch is also passed to the assembler and linker. All modules should be compiled with the same '`-G` *num*' value.

`-nocpp`
: Tell the MIPS assembler to not run its preprocessor over user assembler files (with a '`.s`' suffix) when assembling them.

`-mfix7000`
: Pass an option to gas which will cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

`-no-crt0`
: Do not include the default crt0.

`-mflush-func=`*func*
`-mno-flush-func`
: Specifies the function to call to flush the I and D caches, or to not call any such function. If called, the function must take the same arguments as the

common `_flush_func()`, that is, the address of the memory range for which the cache is being flushed, the size of the memory range, and the number 3 (to flush both caches). The default depends on the target gcc was configured for, but commonly is either '`_flush_func`' or '`__cpu_flush`'.

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

3.17.15 Intel 386 and AMD x86-64 Options

These '`-m`' options are defined for the i386 and x86-64 family of computers:

`-mcpu=`*cpu-type*
: Tune to *cpu-type* everything applicable about the generated code, except for the ABI and the set of available instructions. The choices for *cpu-type* are '`i386`', '`i486`', '`i586`', '`i686`', '`pentium`', '`pentium-mmx`', '`pentiumpro`', '`pentium2`', '`pentium3`', '`pentium4`', '`k6`', '`k6-2`', '`k6-3`', '`athlon`', '`athlon-tbird`', '`athlon-4`', '`athlon-xp`' and '`athlon-mp`'.

  While picking a specific *cpu-type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not run on the i386 without the '`-march=`*cpu-type*' option being used. '`i586`' is equivalent to '`pentium`' and '`i686`' is equivalent to '`pentiumpro`'. '`k6`' and '`athlon`' are the AMD chips as opposed to the Intel ones.

`-march=`*cpu-type*
: Generate instructions for the machine type *cpu-type*. The choices for *cpu-type* are the same as for '`-mcpu`'. Moreover, specifying '`-march=`*cpu-type*' implies '`-mcpu=`*cpu-type*'.

`-m386`
`-m486`
`-mpentium`
`-mpentiumpro`
: These options are synonyms for '`-mcpu=i386`', '`-mcpu=i486`', '`-mcpu=pentium`', and '`-mcpu=pentiumpro`' respectively. These synonyms are deprecated.

`-mfpmath=`*unit*
: generate floating point arithmetics for selected unit *unit*. the choices for *unit* are:

  '`387`'
  : Use the standard 387 floating point coprocessor present majority of chips and emulated otherwise. Code compiled with this option will run almost everywhere. The temporary results are computed in 80bit precesion instead of precision specified by the type resulting in slightly different results compared to most of other chips. See '`-ffloat-store`' for more detailed description.

    This is the default choice for i386 compiler.

  '`sse`'
  : Use scalar floating point instructions present in the SSE instruction set. This instruction set is supported by Pentium3 and newer chips,

in the AMD line by Athlon-4, Athlon-xp and Athlon-mp chips. The earlier version of SSE instruction set supports only single precision arithmetics, thus the double and extended precision arithmetics is still done using 387. Later version, present only in Pentium4 and the future AMD x86-64 chips supports double precision arithmetics too.

For i387 you need to use '`-march=`*cpu-type*', '`-msse`' or '`-msse2`' switches to enable SSE extensions and make this option effective. For x86-64 compiler, these extensions are enabled by default.

The resulting code should be considerably faster in majority of cases and avoid the numerical instability problems of 387 code, but may break some existing code that expects temporaries to be 80bit.

This is the default choice for x86-64 compiler.

'`sse,387`' Attempt to utilize both instruction sets at once. This effectivly double the amount of available registers and on chips with separate execution units for 387 and SSE the execution resources too. Use this option with care, as it is still experimental, because gcc register allocator does not model separate functional units well resulting in instable performance.

`-masm=`*dialect*
Output asm instructions using selected *dialect*. Supported choices are '`intel`' or '`att`' (the default one).

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

`-msoft-float`
Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

On machines where a function returns floating point results in the 80387 register stack, some floating point opcodes may be emitted even if '`-msoft-float`' is used.

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

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

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

`-mno-fancy-math-387`
: Some 387 emulators do not support the `sin`, `cos` and `sqrt` instructions for the 387. Specify this option to avoid generating those instructions. This option is the default on FreeBSD, OpenBSD and NetBSD. This option is overridden when '`-march`' indicates that the target cpu will always have an FPU and so the instruction will not need emulation. As of revision 2.6.1, these instructions are not generated unless you also use the '`-funsafe-math-optimizations`' switch.

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

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

`-m128bit-long-double`
: Control the size of `long double` type. i386 application binary interface specify the size to be 12 bytes, while modern architectures (Pentium and newer) prefer `long double` aligned to 8 or 16 byte boundary. This is impossible to reach with 12 byte long doubles in the array accesses.

  Warning: if you use the '`-m128bit-long-double`' switch, the structures and arrays containing `long double` will change their size as well as function calling convention for function taking `long double` will be modified.

`-m96bit-long-double`
: Set the size of `long double` to 96 bits as required by the i386 application binary interface. This is the default.

`-msvr3-shlib`
`-mno-svr3-shlib`
: Control whether GCC places uninitialized local variables into the `bss` or `data` segments. '`-msvr3-shlib`' places them into `bss`. These options are meaningful only on System V Release 3.

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

  You can specify that an individual function is called with this calling sequence with the function attribute '`stdcall`'. You can also override the '`-mrtd`' option by using the function attribute '`cdecl`'. See Section 5.26 [Function Attributes], page 172.

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

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

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

`-mregparm=`*num*
: Control how many registers are used to pass integer arguments. By default, no registers are used to pass arguments, and at most 3 registers can be used. You can control this behavior for a specific function by using the function attribute '`regparm`'. See Section 5.26 [Function Attributes], page 172.

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

`-mpreferred-stack-boundary=`*num*
: Attempt to keep the stack boundary aligned to a 2 raised to *num* byte boundary. If '`-mpreferred-stack-boundary`' is not specified, the default is 4 (16 bytes or 128 bits), except when optimizing for code size ('`-Os`'), in which case the default is the minimum correct alignment (4 bytes for x86, and 8 bytes for x86-64).

  On Pentium and PentiumPro, `double` and `long double` values should be aligned to an 8 byte boundary (see '`-malign-double`') or suffer significant run time performance penalties. On Pentium III, the Streaming SIMD Extension (SSE) data type `__m128` suffers similar penalties if it is not 16 byte aligned.

  To ensure proper alignment of this values on the stack, the stack boundary must be as aligned as that required by any value stored on the stack. Further, every function must be generated such that it keeps the stack aligned. Thus calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary will most likely misalign the stack. It is recommended that libraries that use callbacks always use the default setting.

  This extra alignment does consume extra stack space, and generally increases code size. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to '`-mpreferred-stack-boundary=2`'.

`-mmmx`
`-mno-mmx`

`-msse`
`-mno-sse`

`-msse2`
`-mno-sse2`
`-m3dnow`
`-mno-3dnow`
: These switches enable or disable the use of built-in functions that allow direct access to the MMX, SSE and 3Dnow extensions of the instruction set.

See Section 5.46.1 [X86 Built-in Functions], page 221, for details of the functions enabled and disabled by these switches.

`-mpush-args`
`-mno-push-args`
: Use PUSH operations to store outgoing parameters. This method is shorter and usually equally fast as method using SUB/MOV operations and is enabled by default. In some cases disabling it may improve performance because of improved scheduling and reduced dependencies.

`-maccumulate-outgoing-args`
: If enabled, the maximum amount of space required for outgoing arguments will be computed in the function prologue. This is faster on most modern CPUs because of reduced dependencies, improved scheduling and reduced stack usage when preferred stack boundary is not equal to 2. The drawback is a notable increase in code size. This switch implies '`-mno-push-args`'.

`-mthreads`
: Support thread-safe exception handling on '`Mingw32`'. Code that relies on thread-safe exception handling must compile and link all code with the '`-mthreads`' option. When compiling, '`-mthreads`' defines '`-D_MT`'; when linking, it links in a special thread helper library '`-lmingwthrd`' which cleans up per thread exception handling data.

`-mno-align-stringops`
: Do not align destination of inlined string operations. This switch reduces code size and improves performance in case the destination is already aligned, but gcc don't know about it.

`-minline-all-stringops`
: By default GCC inlines string operations only when destination is known to be aligned at least to 4 byte boundary. This enables more inlining, increase code size, but may improve performance of code that depends on fast memcpy, strlen and memset for short lengths.

`-momit-leaf-frame-pointer`
: Don't keep the frame pointer in a register for leaf functions. This avoids the instructions to save, set up and restore frame pointers and makes an extra register available in leaf functions. The option '`-fomit-frame-pointer`' removes the frame pointer for all functions which might make debugging harder.

These '`-m`' switches are supported in addition to the above on AMD x86-64 processors in 64-bit environments.

`-m32`
`-m64`
: Generate code for a 32-bit or 64-bit environment. The 32-bit environment sets int, long and pointer to 32 bits and generates code that runs on any i386 system. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits and generates code for AMD's x86-64 architecture.

`-mno-red-zone`
: Do not use a so called red zone for x86-64 code. The red zone is mandated by the x86-64 ABI, it is a 128-byte area beyond the location of the stack pointer that

will not be modified by signal or interrupt handlers and therefore can be used for temporary data without adjusting the stack pointer. The flag '-mno-red-zone' disables this red zone.

3.17.16 HPPA Options

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

-march=*architecture-type*
: Generate code for the specified architecture. The choices for *architecture-type* are '1.0' for PA 1.0, '1.1' for PA 1.1, and '2.0' for PA 2.0 processors. Refer to '/usr/lib/sched.models' on an HP-UX system to determine the proper architecture option for your machine. Code compiled for lower numbered architectures will run on higher numbered architectures, but not the other way around.

  PA 2.0 support currently requires gas snapshot 19990413 or later. The next release of binutils (current is 2.9.1) will probably contain PA 2.0 support.

-mpa-risc-1-0
-mpa-risc-1-1
-mpa-risc-2-0
: Synonyms for '-march=1.0', '-march=1.1', and '-march=2.0' respectively.

-mbig-switch
: Generate code suitable for big switch tables. Use this option only if the assembler/linker complain about out of range branches within a switch table.

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

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

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

-mno-space-regs
: Generate code that assumes the target has no space registers. This allows GCC to generate faster indirect calls and use unscaled index address modes.

  Such code is suitable for level 0 PA systems and kernels.

-mfast-indirect-calls
: Generate code that assumes calls never cross space boundaries. This allows GCC to emit code which performs faster indirect calls.

  This option will not work in the presence of shared libraries or nested functions.

`-mlong-load-store`
: Generate 3-instruction load and store sequences as sometimes required by the HP-UX 10 linker. This is equivalent to the '`+k`' option to the HP compilers.

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

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

`-mschedule=`*cpu-type*
: Schedule code according to the constraints for the machine type *cpu-type*. The choices for *cpu-type* are '`700`' '`7100`', '`7100LC`', '`7200`', and '`8000`'. Refer to '`/usr/lib/sched.models`' on an HP-UX system to determine the proper scheduling option for your machine.

`-mlinker-opt`
: Enable the optimization pass in the HPUX linker. Note this makes symbolic debugging impossible. It also triggers a bug in the HPUX 8 and HPUX 9 linkers in which they give bogus error messages when linking some programs.

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

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

3.17.17 Intel 960 Options

These '`-m`' options are defined for the Intel 960 implementations:

`-m`*cpu-type*
: Assume the defaults for the machine type *cpu-type* for some of the other options, including instruction scheduling, floating point support, and addressing modes. The choices for *cpu-type* are '`ka`', '`kb`', '`mc`', '`ca`', '`cf`', '`sa`', and '`sb`'. The default is '`kb`'.

`-mnumerics`
`-msoft-float`
: The '`-mnumerics`' option indicates that the processor does support floating-point instructions. The '`-msoft-float`' option indicates that floating-point support should not be assumed.

`-mleaf-procedures`
`-mno-leaf-procedures`
: Do (or do not) attempt to alter leaf procedures to be callable with the `bal` instruction as well as `call`. This will result in more efficient code for explicit

calls when the `bal` instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers, or using a linker that doesn't support this optimization.

`-mtail-call`
`-mno-tail-call`
: Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete. The default is '`-mno-tail-call`'.

`-mcomplex-addr`
`-mno-complex-addr`
: Assume (or do not assume) that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series, but they definitely are on the C-series. The default is currently '`-mcomplex-addr`' for all processors except the CB and CC.

`-mcode-align`
`-mno-code-align`
: Align code to 8-byte boundaries for faster fetching (or don't bother). Currently turned on by default for C-series implementations only.

`-mic-compat`
`-mic2.0-compat`
`-mic3.0-compat`
: Enable compatibility with iC960 v2.0 or v3.0.

`-masm-compat`
`-mintel-asm`
: Enable compatibility with the iC960 assembler.

`-mstrict-align`
`-mno-strict-align`
: Do not permit (do permit) unaligned accesses.

`-mold-align`
: Enable structure-alignment compatibility with Intel's gcc release version 1.3 (based on gcc 1.37). This option implies '`-mstrict-align`'.

`-mlong-double-64`
: Implement type '`long double`' as 64-bit floating point numbers. Without the option '`long double`' is implemented by 80-bit floating point numbers. The only reason we have it because there is no 128-bit '`long double`' support in '`fp-bit.c`' yet. So it is only useful for people using soft-float targets. Otherwise, we should recommend against use of it.

3.17.18 DEC Alpha Options

These '`-m`' options are defined for the DEC Alpha implementations:

`-mno-soft-float`
`-msoft-float`

Use (do not use) the hardware floating-point instructions for floating-point operations. When '`-msoft-float`' is specified, functions in '`libgcc.a`' will be used to perform floating-point operations. Unless they are replaced by routines that emulate the floating-point operations, or compiled in such a way as to call such emulations routines, these routines will issue floating-point operations. If you are compiling for an Alpha without floating-point operations, you must ensure that the library is built so as not to call them.

Note that Alpha implementations without floating-point operations are required to have floating-point registers.

`-mfp-reg`
`-mno-fp-regs`

Generate code that uses (does not use) the floating-point register set. '`-mno-fp-regs`' implies '`-msoft-float`'. If the floating-point register set is not used, floating point operands are passed in integer registers as if they were integers and floating-point results are passed in `$0` instead of `$f0`. This is a non-standard calling sequence, so any function with a floating-point argument or return value called by code compiled with '`-mno-fp-regs`' must also be compiled with that option.

A typical use of this option is building a kernel that does not use, and hence need not save and restore, any floating-point registers.

`-mieee` The Alpha architecture implements floating-point hardware optimized for maximum performance. It is mostly compliant with the IEEE floating point standard. However, for full compliance, software assistance is required. This option generates code fully IEEE compliant code *except* that the *inexact-flag* is not maintained (see below). If this option is turned on, the preprocessor macro `_IEEE_FP` is defined during compilation. The resulting code is less efficient but is able to correctly support denormalized numbers and exceptional IEEE values such as not-a-number and plus/minus infinity. Other Alpha compilers call this option '`-ieee_with_no_inexact`'.

`-mieee-with-inexact`

This is like '`-mieee`' except the generated code also maintains the IEEE *inexact-flag*. Turning on this option causes the generated code to implement fully-compliant IEEE math. In addition to `_IEEE_FP`, `_IEEE_FP_EXACT` is defined as a preprocessor macro. On some Alpha implementations the resulting code may execute significantly slower than the code generated by default. Since there is very little code that depends on the *inexact-flag*, you should normally not specify this option. Other Alpha compilers call this option '`-ieee_with_inexact`'.

`-mfp-trap-mode=`*trap-mode*

This option controls what floating-point related traps are enabled. Other Alpha compilers call this option '`-fptm` *trap-mode*'. The trap mode can be set to one of four values:

'n' This is the default (normal) setting. The only traps that are enabled are the ones that cannot be disabled in software (e.g., division by zero trap).

'u' In addition to the traps enabled by 'n', underflow traps are enabled as well.

'su' Like 'su', but the instructions are marked to be safe for software completion (see Alpha architecture manual for details).

'sui' Like 'su', but inexact traps are enabled as well.

`-mfp-rounding-mode=`*rounding-mode*
: Selects the IEEE rounding mode. Other Alpha compilers call this option '`-fprm` *rounding-mode*'. The *rounding-mode* can be one of:

'n' Normal IEEE rounding mode. Floating point numbers are rounded towards the nearest machine number or towards the even machine number in case of a tie.

'm' Round towards minus infinity.

'c' Chopped rounding mode. Floating point numbers are rounded towards zero.

'd' Dynamic rounding mode. A field in the floating point control register (*fpcr*, see Alpha architecture reference manual) controls the rounding mode in effect. The C library initializes this register for rounding towards plus infinity. Thus, unless your program modifies the *fpcr*, 'd' corresponds to round towards plus infinity.

`-mtrap-precision=`*trap-precision*
: In the Alpha architecture, floating point traps are imprecise. This means without software assistance it is impossible to recover from a floating trap and program execution normally needs to be terminated. GCC can generate code that can assist operating system trap handlers in determining the exact location that caused a floating point trap. Depending on the requirements of an application, different levels of precisions can be selected:

'p' Program precision. This option is the default and means a trap handler can only identify which program caused a floating point exception.

'f' Function precision. The trap handler can determine the function that caused a floating point exception.

'i' Instruction precision. The trap handler can determine the exact instruction that caused a floating point exception.

Other Alpha compilers provide the equivalent options called '`-scope_safe`' and '`-resumption_safe`'.

`-mieee-conformant`
: This option marks the generated code as IEEE conformant. You must not use this option unless you also specify '`-mtrap-precision=i`' and either

'-mfp-trap-mode=su' or '-mfp-trap-mode=sui'. Its only effect is to emit the line '.eflag 48' in the function prologue of the generated assembly file. Under DEC Unix, this has the effect that IEEE-conformant math library routines will be linked in.

`-mbuild-constants`
: Normally GCC examines a 32- or 64-bit integer constant to see if it can construct it from smaller constants in two or three instructions. If it cannot, it will output the constant as a literal and generate code to load it from the data segment at runtime.

  Use this option to require GCC to construct *all* integer constants using code, even if it takes more instructions (the maximum is six).

  You would typically use this option to build a shared library dynamic loader. Itself a shared library, it must relocate itself in memory before it can find the variables and constants in its own data segment.

`-malpha-as`
`-mgas`
: Select whether to generate code to be assembled by the vendor-supplied assembler ('-malpha-as') or by the GNU assembler '-mgas'.

`-mbwx`
`-mno-bwx`
`-mcix`
`-mno-cix`
`-mfix`
`-mno-fix`
`-mmax`
`-mno-max`
: Indicate whether GCC should generate code to use the optional BWX, CIX, FIX and MAX instruction sets. The default is to use the instruction sets supported by the CPU type specified via '-mcpu=' option or that of the CPU on which GCC was built if none was specified.

`-mfloat-vax`
`-mfloat-ieee`
: Generate code that uses (does not use) VAX F and G floating point arithmetic instead of IEEE single and double precision.

`-mexplicit-relocs`
`-mno-explicit-relocs`
: Older Alpha assemblers provided no way to generate symbol relocations except via assembler macros. Use of these macros does not allow optimial instruction scheduling. GNU binutils as of version 2.12 supports a new syntax that allows the compiler to explicitly mark which relocations should apply to which instructions. This option is mostly useful for debugging, as GCC detects the capabilities of the assembler when it is built and sets the default accordingly.

`-msmall-data`
`-mlarge-data`
: When '-mexplicit-relocs' is in effect, static data is accessed via *gp-relative* relocations. When '-msmall-data' is used, objects 8 bytes long or smaller are

placed in a *small data area* (the `.sdata` and `.sbss` sections) and are accessed via 16-bit relocations off of the `$gp` register. This limits the size of the small data area to 64KB, but allows the variables to be directly accessed via a single instruction.

The default is '`-mlarge-data`'. With this option the data area is limited to just below 2GB. Programs that require more than 2GB of data must use `malloc` or `mmap` to allocate the data in the heap instead of in the program's data segment.

When generating code for shared libraries, '`-fpic`' implies '`-msmall-data`' and '`-fPIC`' implies '`-mlarge-data`'.

`-mcpu=`*cpu_type*

Set the instruction set and instruction scheduling parameters for machine type *cpu_type*. You can specify either the '`EV`' style name or the corresponding chip number. GCC supports scheduling parameters for the EV4, EV5 and EV6 family of processors and will choose the default values for the instruction set from the processor you specify. If you do not specify a processor type, GCC will default to the processor on which the compiler was built.

Supported values for *cpu_type* are

'`ev4`'

'`ev45`'
'`21064`' Schedules as an EV4 and has no instruction set extensions.

'`ev5`'
'`21164`' Schedules as an EV5 and has no instruction set extensions.

'`ev56`'
'`21164a`' Schedules as an EV5 and supports the BWX extension.

'`pca56`'
'`21164pc`'
'`21164PC`' Schedules as an EV5 and supports the BWX and MAX extensions.

'`ev6`'
'`21264`' Schedules as an EV6 and supports the BWX, FIX, and MAX extensions.

'`ev67`'

'`21264a`' Schedules as an EV6 and supports the BWX, CIX, FIX, and MAX extensions.

`-mtune=`*cpu_type*

Set only the instruction scheduling parameters for machine type *cpu_type*. The instruction set is not changed.

`-mmemory-latency=`*time*

Sets the latency the scheduler should assume for typical memory references as seen by the application. This number is highly dependent on the memory access patterns used by the application and the size of the external cache on the machine.

Valid options for *time* are

'*number*' A decimal number representing clock cycles.

'L1'
'L2'
'L3'
'main' The compiler contains estimates of the number of clock cycles for "typical" EV4 & EV5 hardware for the Level 1, 2 & 3 caches (also called Dcache, Scache, and Bcache), as well as to main memory. Note that L3 is only valid for EV5.

3.17.19 DEC Alpha/VMS Options

These '-m' options are defined for the DEC Alpha/VMS implementations:

`-mvms-return-codes`
Return VMS condition codes from main. The default is to return POSIX style condition (e.g. error) codes.

3.17.20 Clipper Options

These '-m' options are defined for the Clipper implementations:

`-mc300` Produce code for a C300 Clipper processor. This is the default.

`-mc400` Produce code for a C400 Clipper processor, i.e. use floating point registers f8–f15.

3.17.21 H8/300 Options

These '-m' options are defined for the H8/300 implementations:

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '-relax'. See section "ld and the H8/300" in *Using ld*, for a fuller description.

`-mh` Generate code for the H8/300H.

`-ms` Generate code for the H8/S.

`-ms2600` Generate code for the H8/S2600. This switch must be used with '-ms'.

`-mint32` Make `int` data 32 bits by default.

`-malign-300`
On the H8/300H and H8/S, use the same alignment rules as for the H8/300. The default for the H8/300H and H8/S is to align longs and floats on 4 byte boundaries. '-malign-300' causes them to be aligned on 2 byte boundaries. This option has no effect on the H8/300.

3.17.22 SH Options

These '-m' options are defined for the SH implementations:

`-m1` Generate code for the SH1.

`-m2` Generate code for the SH2.

`-m3` Generate code for the SH3.

`-m3e` Generate code for the SH3e.

`-m4-nofpu`
: Generate code for the SH4 without a floating-point unit.

`-m4-single-only`
: Generate code for the SH4 with a floating-point unit that only supports single-precision arithmetic.

`-m4-single`
: Generate code for the SH4 assuming the floating-point unit is in single-precision mode by default.

`-m4` Generate code for the SH4.

`-mb` Compile code for the processor in big endian mode.

`-ml` Compile code for the processor in little endian mode.

`-mdalign` Align doubles at 64-bit boundaries. Note that this changes the calling conventions, and thus some functions from the standard C library will not work unless you recompile it first with '-mdalign'.

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '-relax'.

`-mbigtable`
: Use 32-bit offsets in `switch` tables. The default is to use 16-bit offsets.

`-mfmovd` Enable the use of the instruction `fmovd`.

`-mhitachi`
: Comply with the calling conventions defined by Hitachi.

`-mnomacsave`
: Mark the `MAC` register as call-clobbered, even if '-mhitachi' is given.

`-mieee` Increase IEEE-compliance of floating-point code.

`-misize` Dump instruction size and location in the assembly code.

`-mpadstruct`
: This option is deprecated. It pads structures to multiple of 4 bytes, which is incompatible with the SH ABI.

`-mspace` Optimize for space instead of speed. Implied by '-Os'.

`-mprefergot`
: When generating position-independent code, emit function calls using the Global Offset Table instead of the Procedure Linkage Table.

`-musermode`
: Generate a library function call to invalidate instruction cache entries, after fixing up a trampoline. This library function call doesn't assume it can write to the whole memory address space. This is the default when the target is `sh-*-linux*`.

3.17.23 Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

`-G`
: Create a shared object. It is recommended that '`-symbolic`' or '`-shared`' be used instead.

`-Qy`
: Identify the versions of each tool used by the compiler, in a `.ident` assembler directive in the output.

`-Qn`
: Refrain from adding `.ident` directives to the output file (this is the default).

`-YP,`*dirs*
: Search the directories *dirs*, and no others, for libraries specified with '`-l`'.

`-Ym,`*dir*
: Look in the directory *dir* to find the M4 preprocessor. The assembler uses this option.

3.17.24 TMS320C3x/C4x Options

These '`-m`' options are defined for TMS320C3x/C4x implementations:

`-mcpu=`*cpu_type*
: Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`c30`', '`c31`', '`c32`', '`c40`', and '`c44`'. The default is '`c40`' to generate code for the TMS320C40.

`-mbig-memory`
`-mbig`
`-msmall-memory`
`-msmall`
: Generates code for the big or small memory model. The small memory model assumed that all data fits into one 64K word page. At run-time the data page (DP) register must be set to point to the 64K page containing the .bss and .data program sections. The big memory model is the default and requires reloading of the DP register for every direct memory access.

`-mbk`
`-mno-bk`
: Allow (disallow) allocation of general integer operands into the block count register BK.

`-mdb`
`-mno-db`
: Enable (disable) generation of code using decrement and branch, DBcond(D), instructions. This is enabled by default for the C4x. To be on the safe side, this is disabled for the C3x, since the maximum iteration count on the C3x is $2^{23} + 1$ (but who iterates loops more than 2^{23} times on the C3x?). Note that GCC will try to reverse a loop so that it can utilise the decrement and branch

instruction, but will give up if there is more than one memory reference in the loop. Thus a loop where the loop counter is decremented can generate slightly more efficient code, in cases where the RPTB instruction cannot be utilised.

`-mdp-isr-reload`
`-mparanoid`
: Force the DP register to be saved on entry to an interrupt service routine (ISR), reloaded to point to the data section, and restored on exit from the ISR. This should not be required unless someone has violated the small memory model by modifying the DP register, say within an object library.

`-mmpyi`
`-mno-mpyi`
: For the C3x use the 24-bit MPYI instruction for integer multiplies instead of a library call to guarantee 32-bit results. Note that if one of the operands is a constant, then the multiplication will be performed using shifts and adds. If the '`-mmpyi`' option is not specified for the C3x, then squaring operations are performed inline instead of a library call.

`-mfast-fix`
`-mno-fast-fix`
: The C3x/C4x FIX instruction to convert a floating point value to an integer value chooses the nearest integer less than or equal to the floating point value rather than to the nearest integer. Thus if the floating point number is negative, the result will be incorrectly truncated an additional code is necessary to detect and correct this case. This option can be used to disable generation of the additional code required to correct the result.

`-mrptb`
`-mno-rptb`
: Enable (disable) generation of repeat block sequences using the RPTB instruction for zero overhead looping. The RPTB construct is only used for innermost loops that do not call functions or jump across the loop boundaries. There is no advantage having nested RPTB loops due to the overhead required to save and restore the RC, RS, and RE registers. This is enabled by default with '`-O2`'.

`-mrpts=`*count*
`-mno-rpts`
: Enable (disable) the use of the single instruction repeat instruction RPTS. If a repeat block contains a single instruction, and the loop count can be guaranteed to be less than the value *count*, GCC will emit a RPTS instruction instead of a RPTB. If no value is specified, then a RPTS will be emitted even if the loop count cannot be determined at compile time. Note that the repeated instruction following RPTS does not have to be reloaded from memory each iteration, thus freeing up the CPU buses for operands. However, since interrupts are blocked by this instruction, it is disabled by default.

`-mloop-unsigned`
`-mno-loop-unsigned`
: The maximum iteration count when using RPTS and RPTB (and DB on the C40) is $2^{31} + 1$ since these instructions test if the iteration count is negative to

terminate the loop. If the iteration count is unsigned there is a possibility than the $2^{31}+1$ maximum iteration count may be exceeded. This switch allows an unsigned iteration count.

`-mti` Try to emit an assembler syntax that the TI assembler (asm30) is happy with. This also enforces compatibility with the API employed by the TI C3x C compiler. For example, long doubles are passed as structures rather than in floating point registers.

`-mregparm`
`-mmemparm`
Generate code that uses registers (stack) for passing arguments to functions. By default, arguments are passed in registers where possible rather than by pushing arguments on to the stack.

`-mparallel-insns`
`-mno-parallel-insns`
Allow the generation of parallel instructions. This is enabled by default with '`-O2`'.

`-mparallel-mpy`
`-mno-parallel-mpy`
Allow the generation of MPY||ADD and MPY||SUB parallel instructions, provided '`-mparallel-insns`' is also specified. These instructions have tight register constraints which can pessimize the code generation of large functions.

3.17.25 V850 Options

These '`-m`' options are defined for V850 implementations:

`-mlong-calls`
`-mno-long-calls`
Treat all calls as being far away (near). If calls are assumed to be far away, the compiler will always load the functions address up into a register, and call indirect through the pointer.

`-mno-ep`
`-mep` Do not optimize (do optimize) basic blocks that use the same index pointer 4 or more times to copy pointer into the `ep` register, and use the shorter `sld` and `sst` instructions. The '`-mep`' option is on by default if you optimize.

`-mno-prolog-function`
`-mprolog-function`
Do not use (do use) external functions to save and restore registers at the prolog and epilog of a function. The external functions are slower, but use less code space if more than one function saves the same number of registers. The '`-mprolog-function`' option is on by default if you optimize.

`-mspace` Try to make the code as small as possible. At present, this just turns on the '`-mep`' and '`-mprolog-function`' options.

`-mtda=`*n* Put static or global variables whose size is *n* bytes or less into the tiny data area that register `ep` points to. The tiny data area can hold up to 256 bytes in total (128 bytes for byte references).

`-msda=`*n* Put static or global variables whose size is *n* bytes or less into the small data area that register `gp` points to. The small data area can hold up to 64 kilobytes.

`-mzda=`*n* Put static or global variables whose size is *n* bytes or less into the first 32 kilobytes of memory.

`-mv850` Specify that the target processor is the V850.

`-mbig-switch`
Generate code suitable for big switch tables. Use this option only if the assembler/linker complain about out of range branches within a switch table.

3.17.26 ARC Options

These options are defined for ARC implementations:

`-EL` Compile code for little endian mode. This is the default.

`-EB` Compile code for big endian mode.

`-mmangle-cpu`
Prepend the name of the cpu to all public symbol names. In multiple-processor systems, there are many ARC variants with different instruction and register set characteristics. This flag prevents code compiled for one cpu to be linked with code compiled for another. No facility exists for handling variants that are "almost identical". This is an all or nothing option.

`-mcpu=`*cpu*
Compile code for ARC variant *cpu*. Which variants are supported depend on the configuration. All variants support '`-mcpu=base`', this is the default.

`-mtext=`*text-section*
`-mdata=`*data-section*
`-mrodata=`*readonly-data-section*
Put functions, data, and readonly data in *text-section*, *data-section*, and *readonly-data-section* respectively by default. This can be overridden with the `section` attribute. See Section 5.33 [Variable Attributes], page 184.

3.17.27 NS32K Options

These are the '`-m`' options defined for the 32000 series. The default values for these options depends on which style of 32000 was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m32032`
`-m32032` Generate output for a 32032. This is the default when the compiler is configured for 32032 and 32016 based systems.

`-m32332`
`-m32332` Generate output for a 32332. This is the default when the compiler is configured for 32332-based systems.

`-m32532`
`-m32532` Generate output for a 32532. This is the default when the compiler is configured for 32532-based systems.

`-m32081` Generate output containing 32081 instructions for floating point. This is the default for all systems.

`-m32381` Generate output containing 32381 instructions for floating point. This also implies '`-m32081`'. The 32381 is only compatible with the 32332 and 32532 cpus. This is the default for the pc532-netbsd configuration.

`-mmulti-add`
Try and generate multiply-add floating point instructions `polyF` and `dotF`. This option is only available if the '`-m32381`' option is in effect. Using these instructions requires changes to register allocation which generally has a negative impact on performance. This option should only be enabled when compiling code particularly likely to make heavy use of multiply-add instructions.

`-mnomulti-add`
Do not try and generate multiply-add floating point instructions `polyF` and `dotF`. This is the default on all platforms.

`-msoft-float`
Generate output containing library calls for floating point. **Warning:** the requisite libraries may not be available.

`-mnobitfield`
Do not use the bit-field instructions. On some machines it is faster to use shifting and masking operations. This is the default for the pc532.

`-mbitfield`
Do use the bit-field instructions. This is the default for all platforms except the pc532.

`-mrtd` Use a different function-calling convention, in which functions that take a fixed number of arguments return pop their arguments on return with the `ret` instruction.

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

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

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

This option takes its name from the 680x0 `rtd` instruction.

`-mregparam`
Use a different function-calling convention where the first two arguments are passed in registers.

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

`-mnoregparam`
: Do not pass any arguments in registers. This is the default for all targets.

`-msb`
: It is OK to use the sb as an index register which is always loaded with zero. This is the default for the pc532-netbsd target.

`-mnosb`
: The sb register is not available for use or has not been initialized to zero by the run time system. This is the default for all targets except the pc532-netbsd. It is also implied whenever '`-mhimem`' or '`-fpic`' is set.

`-mhimem`
: Many ns32000 series addressing modes use displacements of up to 512MB. If an address is above 512MB then displacements from zero can not be used. This option causes code to be generated which can be loaded above 512MB. This may be useful for operating systems or ROM code.

`-mnohimem`
: Assume code will be loaded in the first 512MB of virtual address space. This is the default for all platforms.

3.17.28 AVR Options

These options are defined for AVR implementations:

`-mmcu=`*mcu*
: Specify ATMEL AVR instruction set or MCU type.

Instruction set avr1 is for the minimal AVR core, not supported by the C compiler, only for assembler programs (MCU types: at90s1200, attiny10, attiny11, attiny12, attiny15, attiny28).

Instruction set avr2 (default) is for the classic AVR core with up to 8K program memory space (MCU types: at90s2313, at90s2323, attiny22, at90s2333, at90s2343, at90s4414, at90s4433, at90s4434, at90s8515, at90c8534, at90s8535).

Instruction set avr3 is for the classic AVR core with up to 128K program memory space (MCU types: atmega103, atmega603, at43usb320, at76c711).

Instruction set avr4 is for the enhanced AVR core with up to 8K program memory space (MCU types: atmega8, atmega83, atmega85).

Instruction set avr5 is for the enhanced AVR core with up to 128K program memory space (MCU types: atmega16, atmega161, atmega163, atmega32, atmega323, atmega64, atmega128, at43usb355, at94k).

`-msize`
: Output instruction sizes to the asm file.

`-minit-stack=`*N*
: Specify the initial stack address, which may be a symbol or numeric value, '`__stack`' is the default.

`-mno-interrupts`
: Generated code is not compatible with hardware interrupts. Code size will be smaller.

`-mcall-prologues`
: Functions prologues/epilogues expanded as call to appropriate subroutines. Code size will be smaller.

`-mno-tablejump`
: Do not generate tablejump insns which sometimes increase code size.

`-mtiny-stack`
: Change only the low 8 bits of the stack pointer.

3.17.29 MCore Options

These are the '-m' options defined for the Motorola M*Core processors.

`-mhardlit`
`-mhardlit`
`-mno-hardlit`
: Inline constants into the code stream if it can be done in two instructions or less.

`-mdiv`
`-mdiv`
`-mno-div`
: Use the divide instruction. (Enabled by default).

`-mrelax-immediate`
`-mrelax-immediate`
`-mno-relax-immediate`
: Allow arbitrary sized immediates in bit operations.

`-mwide-bitfields`
`-mwide-bitfields`
`-mno-wide-bitfields`
: Always treat bit-fields as int-sized.

`-m4byte-functions`
`-m4byte-functions`
`-mno-4byte-functions`
: Force all functions to be aligned to a four byte boundary.

`-mcallgraph-data`
`-mcallgraph-data`
`-mno-callgraph-data`
: Emit callgraph information.

`-mslow-bytes`
`-mslow-bytes`
`-mno-slow-bytes`
: Prefer word access when reading byte quantities.

`-mlittle-endian`
`-mlittle-endian`
`-mbig-endian`
: Generate code for a little endian target.

`-m210`
`-m210`
`-m340`
: Generate code for the 210 processor.

3.17.30 IA-64 Options

These are the '-m' options defined for the Intel IA-64 architecture.

`-mbig-endian`
: Generate code for a big endian target. This is the default for HPUX.

`-mlittle-endian`
: Generate code for a little endian target. This is the default for AIX5 and Linux.

`-mgnu-as`
`-mno-gnu-as`
: Generate (or don't) code for the GNU assembler. This is the default.

`-mgnu-ld`
`-mno-gnu-ld`
: Generate (or don't) code for the GNU linker. This is the default.

`-mno-pic`
: Generate code that does not use a global pointer register. The result is not position independent code, and violates the IA-64 ABI.

`-mvolatile-asm-stop`
`-mno-volatile-asm-stop`
: Generate (or don't) a stop bit immediately before and after volatile asm statements.

`-mb-step`
: Generate code that works around Itanium B step errata.

`-mregister-names`
`-mno-register-names`
: Generate (or don't) 'in', 'loc', and 'out' register names for the stacked registers. This may make assembler output more readable.

`-mno-sdata`
`-msdata`
: Disable (or enable) optimizations that use the small data section. This may be useful for working around optimizer bugs.

`-mconstant-gp`
: Generate code that uses a single constant global pointer value. This is useful when compiling kernel code.

`-mauto-pic`
: Generate code that is self-relocatable. This implies '-mconstant-gp'. This is useful when compiling firmware code.

`-minline-divide-min-latency`
: Generate code for inline divides using the minimum latency algorithm.

`-minline-divide-max-throughput`
: Generate code for inline divides using the maximum throughput algorithm.

`-mno-dwarf2-asm`
`-mdwarf2-asm`
: Don't (or do) generate assembler code for the DWARF2 line number debugging info. This may be useful when not using the GNU assembler.

`-mfixed-range=`*register-range*
: Generate code treating the given register range as fixed registers. A fixed register is one that the register allocator can not use. This is useful when compiling kernel code. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma.

3.17.31 D30V Options

These '`-m`' options are defined for D30V implementations:

`-mextmem`
: Link the '`.text`', '`.data`', '`.bss`', '`.strings`', '`.rodata`', '`.rodata1`', '`.data1`' sections into external memory, which starts at location `0x80000000`.

`-mextmemory`
: Same as the '`-mextmem`' switch.

`-monchip`
: Link the '`.text`' section into onchip text memory, which starts at location `0x0`. Also link '`.data`', '`.bss`', '`.strings`', '`.rodata`', '`.rodata1`', '`.data1`' sections into onchip data memory, which starts at location `0x20000000`.

`-mno-asm-optimize`
`-masm-optimize`
: Disable (enable) passing '`-O`' to the assembler when optimizing. The assembler uses the '`-O`' option to automatically parallelize adjacent short instructions where possible.

`-mbranch-cost=`*n*
: Increase the internal costs of branches to *n*. Higher costs means that the compiler will issue more instructions to avoid doing a branch. The default is 2.

`-mcond-exec=`*n*
: Specify the maximum number of conditionally executed instructions that replace a branch. The default is 4.

3.17.32 S/390 and zSeries Options

These are the '`-m`' options defined for the S/390 and zSeries architecture.

`-mhard-float`
`-msoft-float`
: Use (do not use) the hardware floating-point instructions and registers for floating-point operations. When '`-msoft-float`' is specified, functions in '`libgcc.a`' will be used to perform floating-point operations. When '`-mhard-float`' is specified, the compiler generates IEEE floating-point instructions. This is the default.

`-mbackchain`
`-mno-backchain`
: Generate (or do not generate) code which maintains an explicit backchain within the stack frame that points to the caller's frame. This is currently needed to allow debugging. The default is to generate the backchain.

`-msmall-exec`
`-mno-small-exec`
Generate (or do not generate) code using the `bras` instruction to do subroutine calls. This only works reliably if the total executable size does not exceed 64k. The default is to use the `basr` instruction instead, which does not have this limitation.

`-m64`
`-m31`
When '`-m31`' is specified, generate code compliant to the Linux for S/390 ABI. When '`-m64`' is specified, generate code compliant to the Linux for zSeries ABI. This allows GCC in particular to generate 64-bit instructions. For the '`s390`' targets, the default is '`-m31`', while the '`s390x`' targets default to '`-m64`'.

`-mmvcle`
`-mno-mvcle`
Generate (or do not generate) code using the `mvcle` instruction to perform block moves. When '`-mno-mvcle`' is specifed, use a `mvc` loop instead. This is the default.

`-mdebug`
`-mno-debug`
Print (or do not print) additional debug information when compiling. The default is to not print debug information.

3.17.33 CRIS Options

These options are defined specifically for the CRIS ports.

`-march=`*architecture-type*
`-mcpu=`*architecture-type*
Generate code for the specified architecture. The choices for *architecture-type* are '`v3`', '`v8`' and '`v10`' for respectively ETRAX 4, ETRAX 100, and ETRAX 100 LX. Default is '`v0`' except for cris-axis-linux-gnu, where the default is '`v10`'.

`-mtune=`*architecture-type*
Tune to *architecture-type* everything applicable about the generated code, except for the ABI and the set of available instructions. The choices for *architecture-type* are the same as for '`-march=`*architecture-type*'.

`-mmax-stack-frame=`*n*
Warn when the stack frame of a function exceeds *n* bytes.

`-melinux-stacksize=`*n*
Only available with the '`cris-axis-aout`' target. Arranges for indications in the program to the kernel loader that the stack of the program should be set to *n* bytes.

`-metrax4`
`-metrax100`
The options '`-metrax4`' and '`-metrax100`' are synonyms for '`-march=v3`' and '`-march=v8`' respectively.

-mpdebug Enable CRIS-specific verbose debug-related information in the assembly code. This option also has the effect to turn off the '#NO_APP' formatted-code indicator to the assembler at the beginning of the assembly file.

-mcc-init
: Do not use condition-code results from previous instruction; always emit compare and test instructions before use of condition codes.

-mno-side-effects
: Do not emit instructions with side-effects in addressing modes other than post-increment.

-mstack-align
-mno-stack-align
-mdata-align
-mno-data-align
-mconst-align
-mno-const-align
: These options (no-options) arranges (eliminate arrangements) for the stack-frame, individual data and constants to be aligned for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by these options.

-m32-bit
-m16-bit
-m8-bit Similar to the stack- data- and const-align options above, these options arrange for stack-frame, writable data and constants to all be 32-bit, 16-bit or 8-bit aligned. The default is 32-bit alignment.

-mno-prologue-epilogue
-mprologue-epilogue
: With '-mno-prologue-epilogue', the normal function prologue and epilogue that sets up the stack-frame are omitted and no return instructions or return sequences are generated in the code. Use this option only together with visual inspection of the compiled code: no warnings or errors are generated when call-saved registers must be saved, or storage for local variable needs to be allocated.

-mno-gotplt
-mgotplt With '-fpic' and '-fPIC', don't generate (do generate) instruction sequences that load addresses for functions from the PLT part of the GOT rather than (traditional on other architectures) calls to the PLT. The default is '-mgotplt'.

-maout Legacy no-op option only recognized with the cris-axis-aout target.

-melf Legacy no-op option only recognized with the cris-axis-elf and cris-axis-linux-gnu targets.

-melinux Only recognized with the cris-axis-aout target, where it selects a GNU/linux-like multilib, include files and instruction set for '-march=v8'.

-mlinux Legacy no-op option only recognized with the cris-axis-linux-gnu target.

`-sim` This option, recognized for the cris-axis-aout and cris-axis-elf arranges to link with input-output functions from a simulator library. Code, initialized data and zero-initialized data are allocated consecutively.

`-sim2` Like '`-sim`', but pass linker options to locate initialized data at 0x40000000 and zero-initialized data at 0x80000000.

3.17.34 MMIX Options

These options are defined for the MMIX:

`-mlibfuncs`
`-mno-libfuncs`
: Specify that intrinsic library functions are being compiled, passing all values in registers, no matter the size.

`-mepsilon`
`-mno-epsilon`
: Generate floating-point comparison instructions that compare with respect to the `rE` epsilon register.

`-mabi=mmixware`
`-mabi=gnu`
: Generate code that passes function parameters and return values that (in the called function) are seen as registers `$0` and up, as opposed to the GNU ABI which uses global registers `$231` and up.

`-mzero-extend`
`-mno-zero-extend`
: When reading data from memory in sizes shorter than 64 bits, use (do not use) zero-extending load instructions by default, rather than sign-extending ones.

`-mknuthdiv`
`-mno-knuthdiv`
: Make the result of a division yielding a remainder have the same sign as the divisor. With the default, '`-mno-knuthdiv`', the sign of the remainder follows the sign of the dividend. Both methods are arithmetically valid, the latter being almost exclusively used.

`-mtoplevel-symbols`
`-mno-toplevel-symbols`
: Prepend (do not prepend) a '`:`' to all global symbols, so the assembly code can be used with the `PREFIX` assembly directive.

`-melf` Generate an executable in the ELF format, rather than the default '`mmo`' format used by the `mmix` simulator.

`-mbranch-predict`
`-mno-branch-predict`
: Use (do not use) the probable-branch instructions, when static branch prediction indicates a probable branch.

`-mbase-addresses`
`-mno-base-addresses`
: Generate (do not generate) code that uses *base addresses*. Using a base address automatically generates a request (handled by the assembler and the linker) for a constant to be set up in a global register. The register is used for one or more base address requests within the range 0 to 255 from the value held in the register. The generally leads to short and fast code, but the number of different data items that can be addressed is limited. This means that a program that uses lots of static data may require '`-mno-base-addresses`'.

3.17.35 PDP-11 Options

These options are defined for the PDP-11:

`-mfpu` Use hardware FPP floating point. This is the default. (FIS floating point on the PDP-11/40 is not supported.)

`-msoft-float`
: Do not use hardware floating point.

`-mac0` Return floating-point results in ac0 (fr0 in Unix assembler syntax).

`-mno-ac0` Return floating-point results in memory. This is the default.

`-m40` Generate code for a PDP-11/40.

`-m45` Generate code for a PDP-11/45. This is the default.

`-m10` Generate code for a PDP-11/10.

`-mbcopy-builtin`
: Use inline `movstrhi` patterns for copying memory. This is the default.

`-mbcopy` Do not use inline `movstrhi` patterns for copying memory.

`-mint16`
`-mno-int32`
: Use 16-bit `int`. This is the default.

`-mint32`
`-mno-int16`
: Use 32-bit `int`.

`-mfloat64`
`-mno-float32`
: Use 64-bit `float`. This is the default.

`-mfloat32`
`-mno-float64`
: Use 32-bit `float`.

`-mabshi` Use `abshi2` pattern. This is the default.

`-mno-abshi`
: Do not use `abshi2` pattern.

`-mbranch-expensive`
: Pretend that branches are expensive. This is for experimenting with code generation only.

`-mbranch-cheap`
: Do not pretend that branches are expensive. This is the default.

`-msplit`
: Generate code for a system with split I&D.

`-mno-split`
: Generate code for a system without split I&D. This is the default.

`-munix-asm`
: Use Unix assembler syntax. This is the default when configured for '`pdp11-*-bsd`'.

`-mdec-asm`
: Use DEC assembler syntax. This is the default when configured for any PDP-11 target other than '`pdp11-*-bsd`'.

3.17.36 Xstormy16 Options

These options are defined for Xstormy16:

`-msim`
: Choose startup files and linker script suitable for the simulator.

3.17.37 Xtensa Options

The Xtensa architecture is designed to support many different configurations. The compiler's default options can be set to match a particular Xtensa configuration by copying a configuration file into the GCC sources when building GCC. The options below may be used to override the default options.

`-mbig-endian`
`-mlittle-endian`
: Specify big-endian or little-endian byte ordering for the target Xtensa processor.

`-mdensity`
`-mno-density`
: Enable or disable use of the optional Xtensa code density instructions.

`-mmac16`
`-mno-mac16`
: Enable or disable use of the Xtensa MAC16 option. When enabled, GCC will generate MAC16 instructions from standard C code, with the limitation that it will use neither the MR register file nor any instruction that operates on the MR registers. When this option is disabled, GCC will translate 16-bit multiply/accumulate operations to a combination of core instructions and library calls, depending on whether any other multiplier options are enabled.

`-mmul16`
`-mno-mul16`
: Enable or disable use of the 16-bit integer multiplier option. When enabled, the compiler will generate 16-bit multiply instructions for multiplications of 16

bits or smaller in standard C code. When this option is disabled, the compiler will either use 32-bit multiply or MAC16 instructions if they are available or generate library calls to perform the multiply operations using shifts and adds.

`-mmul32`
`-mno-mul32`
: Enable or disable use of the 32-bit integer multiplier option. When enabled, the compiler will generate 32-bit multiply instructions for multiplications of 32 bits or smaller in standard C code. When this option is disabled, the compiler will generate library calls to perform the multiply operations using either shifts and adds or 16-bit multiply instructions if they are available.

`-mnsa`
`-mno-nsa`
: Enable or disable use of the optional normalization shift amount (`NSA`) instructions to implement the built-in `ffs` function.

`-mminmax`
`-mno-minmax`
: Enable or disable use of the optional minimum and maximum value instructions.

`-msext`
`-mno-sext`
: Enable or disable use of the optional sign extend (`SEXT`) instruction.

`-mbooleans`
`-mno-booleans`
: Enable or disable support for the boolean register file used by Xtensa coprocessors. This is not typically useful by itself but may be required for other options that make use of the boolean registers (e.g., the floating-point option).

`-mhard-float`
`-msoft-float`
: Enable or disable use of the floating-point option. When enabled, GCC generates floating-point instructions for 32-bit `float` operations. When this option is disabled, GCC generates library calls to emulate 32-bit floating-point operations using integer instructions. Regardless of this option, 64-bit `double` operations are always emulated with calls to library functions.

`-mfused-madd`
`-mno-fused-madd`
: Enable or disable use of fused multiply/add and multiply/subtract instructions in the floating-point option. This has no effect if the floating-point option is not also enabled. Disabling fused multiply/add and multiply/subtract instructions forces the compiler to use separate instructions for the multiply and add/subtract operations. This may be desirable in some cases where strict IEEE 754-compliant results are required: the fused multiply add/subtract instructions do not round the intermediate result, thereby producing results with *more* bits of precision than specified by the IEEE standard. Disabling fused multiply add/subtract instructions also ensures that the program output is not sensitive to the compiler's ability to combine multiply and add/subtract operations.

`-mserialize-volatile`
`-mno-serialize-volatile`

When this option is enabled, GCC inserts `MEMW` instructions before `volatile` memory references to guarantee sequential consistency. The default is '`-mserialize-volatile`'. Use '`-mno-serialize-volatile`' to omit the `MEMW` instructions.

`-mtext-section-literals`
`-mno-text-section-literals`

Control the treatment of literal pools. The default is '`-mno-text-section-literals`', which places literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and it also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. With '`-mtext-section-literals`', the literals are interspersed in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files.

`-mtarget-align`
`-mno-target-align`

When this option is enabled, GCC instructs the assembler to automatically align instructions to reduce branch penalties at the expense of some code density. The assembler attempts to widen density instructions to align branch targets and the instructions following call instructions. If there are not enough preceding safe density instructions to align a target, no widening will be performed. The default is '`-mtarget-align`'. These options do not affect the treatment of auto-aligned instructions like `LOOP`, which the assembler will always align, either by widening density instructions or by inserting no-op instructions.

`-mlongcalls`
`-mno-longcalls`

When this option is enabled, GCC instructs the assembler to translate direct calls to indirect calls unless it can determine that the target of a direct call is in the range allowed by the call instruction. This translation typically occurs for calls to functions in other source files. Specifically, the assembler translates a direct `CALL` instruction into an `L32R` followed by a `CALLX` instruction. The default is '`-mno-longcalls`'. This option should be used in programs where the call target can potentially be out of range. This option is implemented in the assembler, not the compiler, so the assembly code generated by GCC will still show direct call instructions—look at the disassembled object code to see the actual instructions. Note that the assembler will use an indirect call for every cross-file call, not just those that really will be out of range.

3.18 Options for Code Generation Conventions

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

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

`-fexceptions`
Enable exception handling. Generates extra code needed to propagate exceptions. For some targets, this implies GCC will generate frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution. If you do not specify this option, GCC will enable it by default for languages like C++ which normally require exception handling, and disable it for languages like C that do not normally require it. However, you may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in C++. You may also wish to disable this option if you are compiling older C++ programs that don't use exception handling.

`-fnon-call-exceptions`
Generate code that allows trapping instructions to throw exceptions. Note that this requires platform-specific runtime support that does not exist everywhere. Moreover, it only allows *trapping* instructions to throw exceptions, i.e. memory references or floating point instructions. It does not allow exceptions to be thrown from arbitrary signal handlers such as `SIGALRM`.

`-funwind-tables`
Similar to '`-fexceptions`', except that it will just generate any needed static data, but will not affect the generated code in any other way. You will normally not enable this option; instead, a language processor that needs this handling would enable it on your behalf.

`-fasynchronous-unwind-tables`
Generate unwind table in dwarf2 format, if supported by target machine. The table is exact at each instruction boundary, so it can be used for stack unwinding from asynchronous events (such as debugger or garbage collector).

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

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

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

`-freg-struct-return`
Return `struct` and `union` values in registers when possible. This is more efficient for small structures than '`-fpcc-struct-return`'.

If you specify neither '`-fpcc-struct-return`' nor '`-freg-struct-return`', GCC defaults to whichever convention is standard for the target. If there is no standard convention, GCC defaults to '`-fpcc-struct-return`', except on targets where GCC is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

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

`-fshort-double`
: Use the same size for `double` as for `float`.

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

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

`-fno-ident`
: Ignore the '`#ident`' directive.

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

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

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

  '`-fno-verbose-asm`', the default, causes the extra information to be omitted and is useful when comparing two assembler files.

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

`-fvolatile-global`
: Consider all memory references to extern and global data items to be volatile. GCC does not consider static data items to be volatile because of this switch.

`-fvolatile-static`
: Consider all memory references to static data to be volatile.

`-fpic`
: Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of GCC; it is part of the operating system). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that '`-fpic`' does not work; in that case, recompile with '`-fPIC`' instead. (These maximums are 16k on the m88k, 8k on the Sparc, and 32k on the m68k and RS/6000. The 386 has no such limit.)

  Position-independent code requires special support, and therefore works only on certain machines. For the 386, GCC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

`-fPIC`
: If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, m88k, and the Sparc.

  Position-independent code requires special support, and therefore works only on certain machines.

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

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

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

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

  It is an error to used this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results.

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

`-fcall-saved-`*reg*
: Treat the register named *reg* as an allocable 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.

It is an error to used this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results.

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

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

`-fpack-struct`

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

`-finstrument-functions`

Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site. (On some platforms, `__builtin_return_address` does not work beyond the current function, so the call site information may not be available to the profiling functions otherwise.)

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table.

This instrumentation is also done for functions expanded inline in other functions. The profiling calls will indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use '`extern inline`' in your C code, an addressable version of such functions must be provided. (This is normally the case anyways, but if you get lucky and the optimizer always expands the functions inline, you might have gotten away without providing static copies.)

A function may be given the attribute `no_instrument_function`, in which case this instrumentation will not be done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory).

`-fstack-check`

Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

Note that this switch does not actually cause checking to be done; the operating system must do that. The switch causes generation of code to ensure that the operating system sees the stack being extended.

`-fstack-limit-register=`*reg*
`-fstack-limit-symbol=`*sym*
`-fno-stack-limit`

Generate code to ensure that the stack does not grow beyond a certain value, either the value of a register or the address of a symbol. If the stack would grow beyond the value, a signal is raised. For most targets, the signal is raised before the stack overruns the boundary, so it is possible to catch the signal without taking special precautions.

For instance, if the stack starts at absolute address '`0x80000000`' and grows downwards, you can use the flags '`-fstack-limit-symbol=__stack_limit`' and '`-Wl,--defsym,__stack_limit=0x7ffe0000`' to enforce a stack limit of 128KB. Note that this may only work with the GNU linker.

`-fargument-alias`
`-fargument-noalias`
`-fargument-noalias-global`

Specify the possible relationships among parameters and between parameters and global data.

'`-fargument-alias`' specifies that arguments (parameters) may alias each other and may alias global storage.
'`-fargument-noalias`' specifies that arguments do not alias each other, but may alias global storage.
'`-fargument-noalias-global`' specifies that arguments do not alias each other and do not alias global storage.

Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.

`-fleading-underscore`

This option and its counterpart, '`-fno-leading-underscore`', forcibly change the way C symbols are represented in the object file. One use is to help link with legacy assembly code.

Be warned that you should know what you are doing when invoking this option, and that not all targets provide complete support for it.

3.19 Environment Variables Affecting GCC

This section describes several environment variables that affect how GCC operates. Some of them work by specifying directories or prefixes to use when searching for various kinds of files. Some are used to specify other aspects of the compilation environment.

Note that you can also specify places to search using options such as '`-B`', '`-I`' and '`-L`' (see Section 3.14 [Directory Options], page 71). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GCC. See section "Controlling the Compilation Driver '`gcc`'" in *GNU Compiler Collection (GCC) Internals*.

`LANG`
`LC_CTYPE`
`LC_MESSAGES`
`LC_ALL`
: These environment variables control the way that GCC uses localization information that allow GCC to work with different national conventions. GCC inspects the locale categories `LC_CTYPE` and `LC_MESSAGES` if it has been configured to do so. These locale categories can be set to any value supported by your installation. A typical value is '`en_UK`' for English in the United Kingdom.

  The `LC_CTYPE` environment variable specifies character classification. GCC uses it to determine the character boundaries in a string; this is needed for some multibyte encodings that contain quote and escape characters that would otherwise be interpreted as a string end or escape.

  The `LC_MESSAGES` environment variable specifies the language to use in diagnostic messages.

  If the `LC_ALL` environment variable is set, it overrides the value of `LC_CTYPE` and `LC_MESSAGES`; otherwise, `LC_CTYPE` and `LC_MESSAGES` default to the value of the `LANG` environment variable. If none of these variables are set, GCC defaults to traditional C English behavior.

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

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

  If `GCC_EXEC_PREFIX` is not set, GCC will attempt to figure out an appropriate prefix to use based on the pathname it was invoked with.

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

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

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

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

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

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

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

LANG
: This variable is used to pass locale information to the compiler. One way in which this information is used is to determine the character set to be used when character literals, string literals and comments are parsed in C and C++. When the compiler is configured to allow multibyte characters, the following values for LANG are recognized:

  'C-JIS' Recognize JIS characters.

  'C-SJIS' Recognize SJIS characters.

  'C-EUCJP' Recognize EUCJP characters.

  If LANG is not defined, or if it has some other value, then the compiler will use mblen and mbtowc as defined by the default locale to recognize and translate multibyte characters.

Some additional environments variables affect the behavior of the preprocessor.

CPATH
C_INCLUDE_PATH
CPLUS_INCLUDE_PATH
OBJC_INCLUDE_PATH
: Each variable's value is a list of directories separated by a special character, much like PATH, in which to look for header files. The special character, PATH_SEPARATOR, is target-dependent and determined at GCC build time. For Windows-based targets it is a semicolon, and for almost all other targets it is a colon.

  CPATH specifies a list of directories to be searched as if specified with '-I', but after any paths given with '-I' options on the command line. The environment variable is used regardless of which language is being preprocessed.

  The remaining environment variables apply only when preprocessing the particular language indicated. Each specifies a list of directories to be searched as if specified with '-isystem', but after any paths given with '-isystem' options on the command line.

DEPENDENCIES_OUTPUT
: If this variable is set, its value specifies how to output dependencies for Make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.

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

In other words, this environment variable is equivalent to combining the options '`-MM`' and '`-MF`' (see Section 3.11 [Preprocessor Options], page 61), with an optional '`-MT`' switch too.

`SUNPRO_DEPENDENCIES`

This variable is the same as the environment variable `DEPENDENCIES_OUTPUT` (see [DEPENDENCIES_OUTPUT], page 147), except that system header files are not ignored, so it implies '`-M`' rather than '`-MM`'. See Section 3.11 [Preprocessor Options], page 61.

3.20 Running Protoize

The program `protoize` is an optional part of GCC. You can use it to add prototypes to a program, thus converting the program to ISO C in one respect. The companion program `unprotoize` does the reverse: it removes argument types from any prototypes that are found.

When you run these programs, you must specify a set of source files as command line arguments. The conversion programs start out by compiling these files to see what functions they define. The information gathered about a file *foo* is saved in a file named '*foo*`.X`'.

After scanning comes actual conversion. The specified files are all eligible to be converted; any files they include (whether sources or just headers) are eligible as well.

But not all the eligible files are converted. By default, `protoize` and `unprotoize` convert only source and header files in the current directory. You can specify additional directories whose files should be converted with the '`-d` *directory*' option. You can also specify particular files to exclude with the '`-x` *file*' option. A file is converted if it is eligible, its directory name matches one of the specified directory names, and its name within the directory has not been excluded.

Basic conversion with `protoize` consists of rewriting most function definitions and function declarations to specify the types of the arguments. The only ones not rewritten are those for varargs functions.

`protoize` optionally inserts prototype declarations at the beginning of the source file, to make them available for any calls that precede the function's definition. Or it can insert prototype declarations with block scope in the blocks where undeclared functions are called.

Basic conversion with `unprotoize` consists of rewriting most function declarations to remove any argument types, and rewriting function definitions to the old-style pre-ISO form.

Both conversion programs print a warning for any function declaration or definition that they can't convert. You can suppress these warnings with '`-q`'.

The output from `protoize` or `unprotoize` replaces the original source file. The original file is renamed to a name ending with '`.save`' (for DOS, the saved filename ends in '`.sav`' without the original '.c' suffix). If the '.save' ('.sav' for DOS) file already exists, then the source file is simply discarded.

`protoize` and `unprotoize` both depend on GCC itself to scan the program and collect information about the functions it uses. So neither of these programs will work until GCC is installed.

Here is a table of the options you can use with `protoize` and `unprotoize`. Each option works with both programs unless otherwise stated.

`-B` *directory*
: Look for the file 'SYSCALLS.c.X' in *directory*, instead of the usual directory (normally '/usr/local/lib'). This file contains prototype information about standard system functions. This option applies only to `protoize`.

`-c` *compilation-options*
: Use *compilation-options* as the options when running `gcc` to produce the '.X' files. The special option '-aux-info' is always passed in addition, to tell `gcc` to write a '.X' file.

  Note that the compilation options must be given as a single argument to `protoize` or `unprotoize`. If you want to specify several `gcc` options, you must quote the entire set of compilation options to make them a single word in the shell.

  There are certain `gcc` arguments that you cannot use, because they would produce the wrong kind of output. These include '-g', '-O', '-c', '-S', and '-o' If you include these in the *compilation-options*, they are ignored.

`-C`
: Rename files to end in '.C' ('.cc' for DOS-based file systems) instead of '.c'. This is convenient if you are converting a C program to C++. This option applies only to `protoize`.

`-g`
: Add explicit global declarations. This means inserting explicit declarations at the beginning of each source file for each function that is called in the file and was not declared. These declarations precede the first function definition that contains a call to an undeclared function. This option applies only to `protoize`.

`-i` *string*
: Indent old-style parameter declarations with the string *string*. This option applies only to `protoize`.

  `unprotoize` converts prototyped function definitions to old-style function definitions, where the arguments are declared between the argument list and the initial '{'. By default, `unprotoize` uses five spaces as the indentation. If you want to indent with just one space instead, use '-i " "'.

`-k`
: Keep the '.X' files. Normally, they are deleted after conversion is finished.

`-l`
: Add explicit local declarations. `protoize` with '-l' inserts a prototype declaration for each function in each block which calls the function without any declaration. This option applies only to `protoize`.

`-n`
: Make no real changes. This mode just prints information about the conversions that would have been done without '-n'.

`-N` Make no '`.save`' files. The original files are simply deleted. Use this option with caution.

`-p` *program*
: Use the program *program* as the compiler. Normally, the name '`gcc`' is used.

`-q` Work quietly. Most warnings are suppressed.

`-v` Print the version number, just like '`-v`' for `gcc`.

If you need special compiler options to compile one of your program's source files, then you should generate that file's '`.X`' file specially, by running `gcc` on that source file with the appropriate options and the option '`-aux-info`'. Then run `protoize` on the entire set of files. `protoize` will use the existing '`.X`' file because it is newer than the source file. For example:

```
gcc -Dfoo=bar file1.c -aux-info file1.X
protoize *.c
```

You need to include the special files along with the rest in the `protoize` command, even though their '`.X`' files already exist, because otherwise they won't get converted.

See Section 9.10 [Protoize Caveats], page 286, for more information on how to use `protoize` successfully.

4 C Implementation-defined behavior

A conforming implementation of ISO C is required to document its choice of behavior in each of the areas that are designated "implementation defined." The following lists all such areas, along with the section number from the ISO/IEC 9899:1999 standard.

4.1 Translation

- *How a diagnostic is identified (3.10, 5.1.1.3).*
- *Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).*

4.2 Environment

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

4.3 Identifiers

- *Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).*
- *The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).*

4.4 Characters

- *The number of bits in a byte (3.6).*
- *The values of the members of the execution character set (5.2.1).*
- *The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).*
- *The value of a* `char` *object into which has been stored any character other than a member of the basic execution character set (6.2.5).*
- *Which of* `signed char` *or* `unsigned char` *has the same range, representation, and behavior as "plain"* `char` *(6.2.5, 6.3.1.1).*
- *The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).*
- *The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).*
- *The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).*
- *The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).*

- *The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).*
- *The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).*

4.5 Integers

- *Any extended integer types that exist in the implementation (6.2.5).*
- *Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).*
- *The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).*
- *The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).*
- *The results of some bitwise operations on signed integers (6.5).*

4.6 Floating point

- *The accuracy of the floating-point operations and of the library functions in* `<math.h>` *and* `<complex.h>` *that return floating-point results (5.2.4.2.2).*
- *The rounding behaviors characterized by non-standard values of* `FLT_ROUNDS` *(5.2.4.2.2).*
- *The evaluation methods characterized by non-standard negative values of* `FLT_EVAL_METHOD` *(5.2.4.2.2).*
- *The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).*
- *The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).*
- *How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).*
- *Whether and how floating expressions are contracted when not disallowed by the* `FP_CONTRACT` *pragma (6.5).*
- *The default state for the* `FENV_ACCESS` *pragma (7.6.1).*
- *Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).*
- *The default state for the* `FP_CONTRACT` *pragma (7.12.2).*
- *Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).*
- *Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).*

4.7 Arrays and pointers

- *The result of converting a pointer to an integer or vice versa (6.3.2.3).*

  A cast from pointer to integer discards most-significant bits if the pointer representation is larger than the integer type, sign-extends[1] if the pointer representation is smaller than the integer type, otherwise the bits are unchanged.

  A cast from integer to pointer discards most-significant bits if the pointer representation is smaller than the integer type, extends according to the signedness of the integer type if the pointer representation is larger than the integer type, otherwise the bits are unchanged.

  When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in 6.5.6/8.
- *The size of the result of subtracting two pointers to elements of the same array (6.5.6).*

4.8 Hints

- *The extent to which suggestions made by using the* `register` *storage-class specifier are effective (6.7.1).*
- *The extent to which suggestions made by using the inline function specifier are effective (6.7.4).*

4.9 Structures, unions, enumerations, and bit-fields

- *Whether a "plain" int bit-field is treated as a* `signed int` *bit-field or as an* `unsigned int` *bit-field (6.7.2, 6.7.2.1).*
- *Allowable bit-field types other than* `_Bool`, `signed int`, *and* `unsigned int` *(6.7.2.1).*
- *Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).*
- *The order of allocation of bit-fields within a unit (6.7.2.1).*
- *The alignment of non-bit-field members of structures (6.7.2.1).*
- *The integer type compatible with each enumerated type (6.7.2.2).*

4.10 Qualifiers

- *What constitutes an access to an object that has volatile-qualified type (6.7.3).*

4.11 Preprocessing directives

- *How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).*
- *Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).*

[1] Future versions of GCC may zero-extend, or use a target-defined `ptr_extend` pattern. Do not rely on sign extension.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).
- The places that are searched for an included '<>' delimited header, and how the places are specified or the header is identified (6.10.2).
- How the named source file is searched for in an included '""' delimited header (6.10.2).
- The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).
- The nesting limit for #include processing (6.10.2).
- Whether the '#' operator inserts a '\' character before the '\' character that begins a universal character name in a character constant or string literal (6.10.3.2).
- The behavior on each recognized non-STDC #pragma directive (6.10.6).
- The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).

4.12 Library functions

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

4.13 Architecture

- The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.18.2, 7.18.3).
- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).
- The value of the result of the sizeof operator (6.5.3.4).

4.14 Locale-specific behavior

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

5 Extensions to the C Language Family

GNU C provides several language features not found in ISO standard C. (The '-pedantic' option directs GCC 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 GCC.

These extensions are available in C and Objective-C. Most of them are also available in C++. See Chapter 6 [Extensions to the C++ Language], page 249, for extensions that apply *only* to C++.

Some features that are in ISO C99 but not C89 or C++ are also, as extensions, accepted by GCC in C89 mode and in C++.

5.1 Statements and Declarations in Expressions

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

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

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

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

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

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

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

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

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

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

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see Section 5.7 [Typeof], page 161) or type naming (see Section 5.6 [Naming Types], page 160).

Statement expressions are not supported fully in G++, and their fate there is unclear. (It is possible that they will become fully supported at some point, or that they will be deprecated, or that the bugs that are present will continue to exist indefinitely.) Presently, statement expressions do not work well as default arguments.

In addition, there are semantic issues with statement-expressions in C++. If you try to use statement-expressions instead of inline functions in C++, you may be surprised at the way object destruction is handled. For example:

```
#define foo(a)  ({int b = (a); b + 3; })
```

does not work the same way as:

```
inline int foo(int a) { int b = a; return b + 3; }
```

In particular, if the expression passed into `foo` involves the creation of temporaries, the destructors for those temporaries will be run earlier in the case of the macro than in the case of the function.

These considerations mean that it is probably a bad idea to use statement-expressions of this form in header files that are designed to work with C++. (Note that some versions of the GNU C Library contained header files using statement-expression that lead to precisely this bug.)

5.2 Locally Declared Labels

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

A local label declaration looks like this:

`__label__` *label*;

or

`__label__` *label1*, *label2*, ...;

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

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

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

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
```

```
        if (_SEARCH_array[i][j] == _SEARCH_target)  \
          { value = i; goto found; }                \
    value = -1;                                     \
   found:                                           \
    value;                                          \
  })
```

5.3 Labels as Values

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

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

To use these values, you need to be able to jump to one. This is done with the computed goto statement[1], `goto *`*exp*`;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

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

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

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

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

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

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

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

You may not use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

An alternate way to write the above example is

```
static const int array[] = { &&foo - &&foo, &&bar - &&foo,
                             &&hack - &&foo };
goto *(&&foo + array[i]);
```

This is more friendly to code living in shared libraries, as it reduces the number of dynamic relocations that are needed, and by consequence, allows the data to be read-only.

---

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

5.4 Nested Functions

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

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

  return square (a) + square (b);
}
```

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

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

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

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

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

  intermediate (store, size);
}
```

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

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

GCC implements taking the address of a nested function using a technique called *trampolines*. A paper describing them is available as

`http://people.debian.org/~karlheg/Usenix88-lexic.pdf`.

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see Section 5.2 [Local Labels], page 156). Such a jump returns instantly to the containing function, exiting the nested function which did the goto and any intermediate functions as well. Here is an example:

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

 /* Control comes here from access
    if it detects an error.  */
 failure:
  return -1;
}
```

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

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

5.5 Constructing Function Calls

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

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

`void * __builtin_apply_args ()` Built-in Function

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

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

`void * __builtin_apply (void (*`*function*`)(), void *`*arguments*`, size_t` *size*`)` Built-in Function

This built-in function invokes *function* with a copy of the parameters described by *arguments* and *size*.

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

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

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

`void __builtin_return (void *`*result*`)` Built-in Function

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

5.6 Naming an Expression's Type

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

```
typedef name = exp;
```

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

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

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

5.7 Referring to a Type with typeof

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

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

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

This assumes that `x` is an array of pointers to functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

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

If you are writing a header file that must work when included in ISO C programs, write `__typeof__` instead of `typeof`. See Section 5.40 [Alternate Keywords], page 213.

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

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

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

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

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

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

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

  It is equivalent to the following traditional C declaration:

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

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

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

  Now the declaration can be rewritten this way:

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

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

5.8 Generalized Lvalues

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.9 Conditionals with Omitted Operands

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

Therefore, the expression

```
x ? : y
```

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

This example is perfectly equivalent to

```
x ? x : y
```

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

5.10 Double-Word Integers

ISO C99 supports data types for integers that are at least 64 bits wide, and as an extension GCC supports them in C89 mode and in C++. Simply write `long long int` for a signed integer, or `unsigned long long int` for an unsigned integer. To make an integer constant of type `long long int`, add the suffix 'LL' to the integer. To make an integer constant of type `unsigned long long int`, add the suffix 'ULL' to the integer.

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

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

5.11 Complex Numbers

ISO C99 supports complex floating data types, and as an extension GCC supports them in C89 mode and in C++, and supports complex integer data types which are not part of ISO C99. You can declare complex types using the keyword `_Complex`. As an extension, the older GNU keyword `__complex__` is also supported.

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

To write a constant with a complex data type, use the suffix '`i`' or '`j`' (either one; they are equivalent). For example, `2.5fi` has type `_Complex float` and `3i` has type `_Complex int`. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant. This is a GNU extension; if you have an ISO C99 conforming C library (such as GNU libc), and want to construct complex constants of floating type, you should include `<complex.h>` and use the macros `I` or `_Complex_I` instead.

To extract the real part of a complex-valued expression *exp*, write `__real__` *exp*. Likewise, use `__imag__` to extract the imaginary part. This is a GNU extension; for values of floating type, you should use the ISO C99 functions `crealf`, `creal`, `creall`, `cimagf`, `cimag` and `cimagl`, declared in `<complex.h>` and also provided as built-in functions by GCC.

The operator '`~`' performs complex conjugation when used on a value with a complex type. This is a GNU extension; for values of floating type, you should use the ISO C99 functions `conjf`, `conj` and `conjl`, declared in `<complex.h>` and also provided as built-in functions by GCC.

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

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

5.12 Hex Floats

ISO C99 supports floating-point numbers written not only in the usual decimal notation, such as `1.55e1`, but also numbers such as `0x1.fp3` written in hexadecimal format. As a GNU extension, GCC supports this in C89 mode (except in some cases when strictly conforming) and in C++. In that format the '`0x`' hex introducer and the '`p`' or '`P`' exponent field are mandatory. The exponent is a decimal number that indicates the power of 2 by which the significant part will be multiplied. Thus '`0x1.f`' is $1\frac{15}{16}$, '`p3`' multiplies it by 8, and the value of `0x1.fp3` is the same as `1.55e1`.

Unlike for floating-point numbers in the decimal notation the exponent is always required in the hexadecimal notation. Otherwise the compiler would not be able to resolve the ambiguity of, e.g., `0x1.f`. This could mean `1.0f` or `1.9375` since '`f`' is also the extension for floating-point constants of type `float`.

5.13 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 ISO C89, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

In ISO C99, you would use a *flexible array member*, which is slightly different in syntax and semantics:

- Flexible array members are written as `contents[]` without the `0`.
- Flexible array members have incomplete type, and so the `sizeof` operator may not be applied. As a quirk of the original implementation of zero-length arrays, `sizeof` evaluates to zero.

- Flexible array members may only appear as the last member of a `struct` that is otherwise non-empty.

GCC versions before 3.0 allowed zero-length arrays to be statically initialized, as if they were flexible arrays. In addition to those cases that were useful, it also allowed initializations in situations that would corrupt later data. Non-empty initialization of zero-length arrays is now treated like any case where there are more initializer elements than the array holds, in that a suitable warning about "excess elements in array" is given, and the excess elements (all of them, in this case) are ignored.

Instead GCC allows static initialization of flexible array members. This is equivalent to defining a new structure containing the original structure followed by an array of sufficient size to contain the data. I.e. in the following, `f1` is constructed as if it were declared like `f2`.

```
struct f1 {
  int x; int y[];
} f1 = { 1, { 2, 3, 4 } };

struct f2 {
  struct f1 f1; int data[3];
} f2 = { { 1 }, { 2, 3, 4 } };
```

The convenience of this extension is that `f1` has the desired type, eliminating the need to consistently refer to `f2.f1`.

This has symmetry with normal static arrays, in that an array of unknown size is also written with `[]`.

Of course, this extension only makes sense if the extra data comes at the end of a top-level object, as otherwise we would be overwriting data at subsequent offsets. To avoid undue complication and confusion with initialization of deeply nested arrays, we simply disallow any non-empty initialization except when the structure is the top-level object. For example:

```
struct foo { int x; int y[]; };
struct bar { struct foo z; };

struct foo a = { 1, { 2, 3, 4 } };        // Valid.
struct bar b = { { 1, { 2, 3, 4 } } };    // Invalid.
struct bar c = { { 1, { } } };            // Valid.
struct foo d[1] = { { 1 { 2, 3, 4 } } };  // Invalid.
```

5.14 Arrays of Variable Length

Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C89 mode and in C++. (However, GCC's implementation of variable-length arrays does not yet conform in detail to the ISO C99 standard.) These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
```

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

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

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

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

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

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

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

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

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

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

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type. ISO C99 does not support parameter forward declarations.

5.15 Macros with a Variable Number of Arguments.

In the ISO C standard of 1999, a macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. Here is an example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

Here '...' is a *variable argument*. In the invocation of such a macro, it represents the zero or more tokens until the closing parenthesis that ends the invocation, including any commas. This set of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. See the CPP manual for more information.

GCC has long supported variadic macros, and used a different syntax that allowed you to give a name to the variable arguments just like any other argument. Here is an example:

```
#define debug(format, args...) fprintf (stderr, format, args)
```

This is in all ways equivalent to the ISO C example above, but arguably more readable and descriptive.

GNU CPP has two further variadic macro extensions, and permits them to be used with either of the above forms of macro definition.

In standard C, you are not allowed to leave the variable argument out entirely; but you are allowed to pass an empty argument. For example, this invocation is invalid in ISO C, because there is no comma after the string:

```
debug ("A message")
```

GNU CPP permits you to completely omit the variable arguments in this way. In the above examples, the compiler would complain, though since the expansion of the macro still has the extra comma after the format string.

To help solve this problem, CPP behaves specially for variable arguments used with the token paste operator, '`##`'. If instead you write

```
#define debug(format, ...) fprintf (stderr, format, ## __VA_ARGS__)
```

and if the variable arguments are omitted or empty, the '`##`' operator causes the preprocessor to remove the comma before it. If you do provide some variable arguments in your macro invocation, GNU CPP does not complain about the paste operation and instead places the variable arguments after the comma. Just like any other pasted macro argument, these arguments are not macro expanded.

5.16 Slightly Looser Rules for Escaped Newlines

Recently, the non-traditional preprocessor has relaxed its treatment of escaped newlines. Previously, the newline had to immediately follow a backslash. The current implementation allows whitespace in the form of spaces, horizontal and vertical tabs, and form feeds between the backslash and the subsequent newline. The preprocessor issues a warning, but treats it as a valid escaped newline and combines the two lines to form a single logical line. This works within comments and tokens, including multi-line strings, as well as between tokens. Comments are *not* treated as whitespace for the purposes of this relaxation, since they have not yet been replaced with spaces.

5.17 String Literals with Embedded Newlines

As an extension, GNU CPP permits string literals to cross multiple lines without escaping the embedded newlines. Each embedded newline is replaced with a single '`\n`' character in the resulting string literal, regardless of what form the newline took originally.

CPP currently allows such strings in directives as well (other than the '#include' family). This is deprecated and will eventually be removed.

5.18 Non-Lvalue Arrays May Have Subscripts

In ISO C99, arrays that are not lvalues still decay to pointers, and may be subscripted, although they may not be modified or used after the next sequence point and the unary '&' operator may not be applied to them. As an extension, GCC allows such arrays to be subscripted in C89 mode, though otherwise they do not decay to pointers outside C99 mode. For example, this is valid in GNU C though not valid in C89:

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

struct foo f();

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

5.19 Arithmetic on `void`- and Function-Pointers

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

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

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

5.20 Non-Constant Initializers

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

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

5.21 Compound Literals

ISO C99 supports compound literals. A compound literal 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; it is an lvalue. As an extension, GCC supports compound literals in C89 mode and in C++.

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

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

Here is an example of constructing a `struct foo` with a compound literal:

```
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 compound literal are (made up of) simple constant expressions, suitable for use in initializers of objects of static storage duration, then the compound literal can be coerced to a pointer to its first element and used in such an initializer, as shown here:

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

Compound literals for scalar types and union types are is also allowed, but then the compound literal is equivalent to a cast.

As a GNU extension, GCC allows initialization of objects with static storage duration by compound literals (which is not possible in ISO C99, because the initializer is not a constant). It is handled as if the object was initialized only with the bracket enclosed list if compound literal's and object types match. The initializer list of the compound literal must be constant. If the object being initialized has array type of unknown size, the size is determined by compound literal size.

```
static struct foo x = (struct foo) {1, 'a', 'b'};
static int y[] = (int []) {1, 2, 3};
static int z[] = (int [3]) {1};
```

The above lines are equivalent to the following:

```
static struct foo x = {1, 'a', 'b'};
static int y[] = {1, 2, 3};
static int z[] = {1, 0, 0};
```

5.22 Designated Initializers

Standard C89 requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In ISO C99 you can give the elements in any order, specifying the array indices or structure field names they apply to, and GNU C allows this as an extension in C89 mode as well. This extension is not implemented in GNU C++.

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

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

is equivalent to

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

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

An alternative syntax for this which has been obsolete since GCC 2.5 but GCC still accepts is to write '`[`*index*`]`' before the element value, with no '`=`'.

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

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

If the value in it has side-effects, the side-effects will happen only once, not for each initialized field by the range initializer.

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

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

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

the following initialization

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

is equivalent to

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

Another syntax which has the same meaning, obsolete since GCC 2.5, is '*fieldname*:', as shown here:

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

The '[*index*]' or '.*fieldname*' is known as a *designator*. You can also use a designator (or the obsolete colon syntax) when initializing a union, to specify which element of the union should be used. For example,

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

union foo f = { .d = 4 };
```

will convert 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer `i`, since it is an integer. (See Section 5.24 [Cast to Union], page 171.)

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

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

is equivalent to

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

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

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

You can also write a series of '.*fieldname*' and '[*index*]' designators before an '=' to specify a nested subobject to initialize; the list is taken relative to the subobject corresponding to the closest surrounding brace pair. For example, with the 'struct point' declaration above:

```
struct point ptarray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

If the same field is initialized multiple times, it will have value from the last initialization. If any such overridden initialization has side-effect, it is unspecified whether the side-effect happens or not. Currently, gcc will discard them and issue a warning.

5.23 Case Ranges

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

```
case low ... high:
```

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

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

```
case 'A' ... 'Z':
```

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

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

rather than this:

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

5.24 Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with `union` *tag* or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See Section 5.21 [Compound Literals], page 168.)

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

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

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

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

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

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

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

5.25 Mixed Declarations and Code

ISO C99 and ISO C++ allow declarations and code to be freely mixed within compound statements. As an extension, GCC also allows this in C89 mode. For example, you could do:

```
int i;
...
i++;
int j = i + 2;
```

Each identifier is visible from where it is declared until the end of the enclosing block.

5.26 Declaring Attributes of Functions

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

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions on all targets: `noreturn`, `noinline`, `always_inline`, `pure`, `const`, `format`, `format_arg`, `no_instrument_function`, `section`, `constructor`, `destructor`, `used`, `unused`, `deprecated`, `weak`, `malloc`, and `alias`. Several other attributes are defined for functions on particular target systems. Other attributes, including `section` are supported for variables declarations (see Section 5.33 [Variable Attributes], page 184) and for types (see Section 5.34 [Type Attributes], page 188).

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

See Section 5.27 [Attribute Syntax], page 179, for details of the exact syntax for using attributes.

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

```
void fatal () __attribute__ ((noreturn));

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

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

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

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

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

```
typedef void voidfn ();

volatile voidfn fatal;
```

`noinline` This function attribute prevents a function from being considered for inlining.

`always_inline`
Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.

`pure` Many functions have no effects except the return value and their return value depends only on the parameters and/or global variables. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `pure`. For example,

```
int square (int) __attribute__ ((pure));
```

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

Some of common examples of pure functions are `strlen` or `memcmp`. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as `feof` in a multithreading environment).

The attribute `pure` is not implemented in GCC versions earlier than 2.96.

`const` Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the `pure` attribute above, since function is not allowed to read global memory.

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

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

```
typedef int intfn ();

extern const intfn square;
```

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

`format (`*archetype*`, `*string-index*`, `*first-to-check*`)`
The `format` attribute specifies that a function takes `printf`, `scanf`, `strftime` or `strfmon` style arguments which should be type-checked against a format string. For example, the declaration:

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

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

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

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

The `format` attribute allows you to identify your own functions which take format strings as arguments, so that GCC can check the calls to these functions for errors. The compiler always (unless '`-ffreestanding`' is used) checks formats for the standard library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `strftime`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using '`-Wformat`'), so there is no need to modify the header file '`stdio.h`'. In C99 mode, the functions `snprintf`, `vsnprintf`, `vscanf`, `vfscanf` and `vsscanf` are also checked. Except in strictly conforming C standard modes, the X/Open function `strfmon` is also checked as are `printf_unlocked` and `fprintf_unlocked`. See Section 3.4 [Options Controlling C Dialect], page 19.

`format_arg (`*string-index*`)`

The `format_arg` attribute specifies that a function takes a format string for a `printf`, `scanf`, `strftime` or `strfmon` style function and modifies it (for example, to translate it into another language), so the result can be passed to a `printf`, `scanf`, `strftime` or `strfmon` style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). For example, the declaration:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

causes the compiler to check the arguments in calls to a `printf`, `scanf`, `strftime` or `strfmon` type function, whose format string argument is a call to the `my_dgettext` function, for consistency with the format string argument `my_format`. If the `format_arg` attribute had not been specified, all the compiler could tell in such calls to format functions would be that the format string argument is not constant; this would generate a warning when '`-Wformat-nonliteral`' is used, but the calls could not be checked without the attribute.

The parameter *string-index* specifies which argument is the format string argument (starting from 1).

The `format-arg` attribute allows you to identify your own functions which modify format strings, so that GCC can check the calls to `printf`, `scanf`, `strftime` or `strfmon` type function whose operands are a call to one of your own function. The compiler always treats `gettext`, `dgettext`, and `dcgettext` in this manner except when strict ISO C support is requested by '`-ansi`' or an appropriate '`-std`' option, or '`-ffreestanding`' is used. See Section 3.4 [Options Controlling C Dialect], page 19.

`no_instrument_function`

If '`-finstrument-functions`' is given, profiling function calls will be generated at entry and exit of most user-compiled functions. Functions with this attribute will not be so instrumented.

`section ("`*section-name*`")`

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

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

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

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

`constructor`
`destructor`

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

These attributes are not currently implemented for Objective-C.

`unused` This attribute, attached to a function, means that the function is meant to be possibly unused. GCC will not produce a warning for this function. GNU C++ does not currently support this attribute as definitions without parameters are valid in C++.

`used` This attribute, attached to a function, means that code must be emitted for the function even if it appears that the function is not referenced. This is useful, for example, when the function is referenced only in inline assembly.

`deprecated`

The `deprecated` attribute results in a warning if the function is used anywhere in the source file. This is useful when identifying functions that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated function, to enable users to easily find further information about why the function is deprecated, or what they should do instead. Note that the warnings only occurs for uses:

```
int old_fn () __attribute__ ((deprecated));
int old_fn ();
int (*fn_ptr)() = old_fn;
```

results in a warning on line 3 but not line 2.

The `deprecated` attribute can also be used for variables and types (see Section 5.33 [Variable Attributes], page 184, see Section 5.34 [Type Attributes], page 188.)

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

`malloc` The `malloc` attribute is used to tell the compiler that a function may be treated as if it were the malloc function. The compiler assumes that calls to malloc result in a pointers that cannot alias anything. This will often improve optimization.

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

```
void __f () { /* do something */; }
void f () __attribute__ ((weak, alias ("__f")));
```

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

Not all target machines support this attribute.

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

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

The PowerPC compiler for Windows NT currently ignores the `stdcall` attribute.

`cdecl` On the Intel 386, the `cdecl` attribute causes the compiler to assume that the calling function will pop off the stack space used to pass arguments. This is useful to override the effects of the '`-mrtd`' switch.

The PowerPC compiler for Windows NT currently ignores the `cdecl` attribute.

`longcall` On the RS/6000 and PowerPC, the `longcall` attribute causes the compiler to always call the function via a pointer, so that functions which reside further than 64 megabytes (67,108,864 bytes) from the current location can be called.

`long_call/short_call`

This attribute allows to specify how to call a particular function on ARM. Both attributes override the '`-mlong-calls`' (see Section 3.17.7 [ARM Options], page 89) command line switch and `#pragma long_calls` settings. The `long_call` attribute causes the compiler to always call the function by first loading its address into a register and then using the contents of that register. The `short_call` attribute always places the offset to the function from the call site into the '`BL`' instruction directly.

`dllimport`

On the PowerPC running Windows NT, the `dllimport` attribute causes the compiler to call the function via a global pointer to the function pointer that is set up by the Windows NT dll library. The pointer name is formed by combining `__imp_` and the function name.

`dllexport`

On the PowerPC running Windows NT, the `dllexport` attribute causes the compiler to provide a global pointer to the function pointer, so that it can be called with the `dllimport` attribute. The pointer name is formed by combining `__imp_` and the function name.

`exception (`*except-func* `[,` *except-arg*`])`

On the PowerPC running Windows NT, the `exception` attribute causes the compiler to modify the structured exception table entry it emits for the declared function. The string or identifier *except-func* is placed in the third entry of the structured exception table. It represents a function, which is called by the exception handling mechanism if an exception occurs. If it was specified, the string or identifier *except-arg* is placed in the fourth entry of the structured exception table.

`function_vector`

Use this attribute on the H8/300 and H8/300H to indicate that the specified function should be called through the function vector. Calling a function through the function vector will reduce code size, however; the function vector has a limited size (maximum 128 entries on the H8/300 and 64 entries on the H8/300H) and shares space with the interrupt vector.

You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

`interrupt`

Use this attribute on the ARM, AVR, M32R/D and Xstormy16 ports to indicate that the specified function is an interrupt handler. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

Note, interrupt handlers for the H8/300, H8/300H and SH processors can be specified via the `interrupt_handler` attribute.

Note, on the AVR interrupts will be enabled inside the function.

Note, for the ARM you can specify the kind of interrupt to be handled by adding an optional parameter to the interrupt attribute like this:

```
void f () __attribute__ ((interrupt ("IRQ")));
```

Permissible values for this parameter are: IRQ, FIQ, SWI, ABORT and UNDEF.

`interrupt_handler`
: Use this attribute on the H8/300, H8/300H and SH to indicate that the specified function is an interrupt handler. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`sp_switch`
: Use this attribute on the SH to indicate an `interrupt_handler` function should switch to an alternate stack. It expects a string argument that names a global variable holding the address of the alternate stack.

```
void *alt_stack;
void f () __attribute__ ((interrupt_handler,
                          sp_switch ("alt_stack")));
```

`trap_exit`
: Use this attribute on the SH for an `interrupt_handle` to return using `trapa` instead of `rte`. This attribute expects an integer argument specifying the trap number to be used.

`eightbit_data`
: Use this attribute on the H8/300 and H8/300H to indicate that the specified variable should be placed into the eight bit data section. The compiler will generate more efficient code for certain operations on data in the eight bit data area. Note the eight bit data area is limited to 256 bytes of data.

You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

`tiny_data`
: Use this attribute on the H8/300H to indicate that the specified variable should be placed into the tiny data section. The compiler will generate more efficient code for loads and stores on data in the tiny data section. Note the tiny data area is limited to slightly under 32kbytes of data.

`signal`
: Use this attribute on the AVR to indicate that the specified function is an signal handler. The compiler will generate function entry and exit sequences suitable for use in an signal handler when this attribute is present. Interrupts will be disabled inside function.

`naked`
: Use this attribute on the ARM or AVR ports to indicate that the specified function do not need prologue/epilogue sequences generated by the compiler. It is up to the programmer to provide these sequences.

`model (`*model-name*`)`
: Use this attribute on the M32R/D to set the addressability of an object, and the code generated for a function. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and are callable with the `bl` instruction.

Medium model objects may live anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and are callable with the `bl` instruction.

Large model objects may live anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and may not be reachable with the `bl` instruction (the compiler will generate the much slower `seth/add3/jl` instruction sequence).

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

Some people object to the `__attribute__` feature, suggesting that ISO C's `#pragma` should be used instead. At the time `__attribute__` was designed, there were two reasons for not doing this.

1. It is impossible to generate `#pragma` commands from a macro.
2. There is no telling what the same `#pragma` might mean in another compiler.

These two reasons applied to almost any application that might have been proposed for `#pragma`. It was basically a mistake to use `#pragma` for *anything*.

The ISO C99 standard includes `_Pragma`, which now allows pragmas to be generated from macros. In addition, a `#pragma GCC` namespace is now in use for GCC-specific pragmas. However, it has been found convenient to use `__attribute__` to achieve a natural attachment of attributes to their corresponding declarations, whereas `#pragma GCC` is of use for constructs that do not naturally form part of the grammar. See section "Miscellaneous Preprocessing Directives" in *The C Preprocessor*.

5.27 Attribute Syntax

This section describes the syntax with which `__attribute__` may be used, and the constructs to which attribute specifiers bind, for the C language. Some details may vary for C++ and Objective-C. Because of infelicities in the grammar for attributes, some forms described here may not be successfully parsed in all cases.

There are some problems with the semantics of attributes in C++. For example, there are no manglings for attributes, although they may affect code generation, so problems may arise when attributed types are used in conjunction with templates or overloading. Similarly, `typeid` does not distinguish between types with different attributes. Support for attributes in C++ may be restricted in future to attributes on declarations only, but not on nested declarators.

See Section 5.26 [Function Attributes], page 172, for details of the semantics of attributes applying to functions. See Section 5.33 [Variable Attributes], page 184, for details of the semantics of attributes applying to variables. See Section 5.34 [Type Attributes], page 188, for details of the semantics of attributes applying to structure, union and enumerated types.

An *attribute specifier* is of the form `__attribute__ ((`*attribute-list*`))`. An *attribute list* is a possibly empty comma-separated sequence of *attributes*, where each attribute is one of the following:

- Empty. Empty attributes are ignored.
- A word (which may be an identifier such as `unused`, or a reserved word such as `const`).
- A word, followed by, in parentheses, parameters for the attribute. These parameters take one of the following forms:
  - An identifier. For example, `mode` attributes use this form.
  - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, `format` attributes use this form.
  - A possibly empty comma-separated list of expressions. For example, `format_arg` attributes use this form with the list being a single integer constant expression, and `alias` attributes use this form with the list being a single string constant.

An *attribute specifier list* is a sequence of one or more attribute specifiers, not separated by any other tokens.

An attribute specifier list may appear after the colon following a label, other than a `case` or `default` label. The only attribute it makes sense to use after a label is `unused`. This feature is intended for code generated by programs which contains labels that may be unused but which is compiled with '`-Wall`'. It would not normally be appropriate to use in it human-written code, though it could be useful in cases where the code that jumps to the label is contained within an `#ifdef` conditional.

An attribute specifier list may appear as part of a `struct`, `union` or `enum` specifier. It may go either immediately after the `struct`, `union` or `enum` keyword, or after the closing brace. It is ignored if the content of the structure, union or enumerated type is not defined in the specifier in which the attribute specifier list is used—that is, in usages such as `struct __attribute__((foo)) bar` with no following opening brace. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type defined is not complete until after the attribute specifiers.

Otherwise, an attribute specifier appears as part of a declaration, counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration), or to a particular declarator within a declaration. Where an attribute specifier is applied to a parameter declared as a function or an array, it should apply to the function or array rather than the pointer to which the parameter is implicitly converted, but this is not yet correctly implemented.

Any list of specifiers and qualifiers at the start of a declaration may contain attribute specifiers, whether or not such a list may in that context contain storage class specifiers. (Some attributes, however, are essentially in the nature of storage class specifiers, and only make sense where storage class specifiers may be used; for example, `section`.) There is one necessary limitation to this syntax: the first old-style parameter declaration in a function definition cannot begin with an attribute specifier, because such an attribute applies to the function instead by syntax described below (which, however, is not yet implemented in this case). In some other cases, attribute specifiers are permitted by this grammar but not yet supported by the compiler. All attribute specifiers in this place relate to the declaration as a whole. In the obsolescent usage where a type of `int` is implied by the absence of type specifiers, such a list of specifiers and qualifiers may be an attribute specifier list with no other specifiers or qualifiers.

An attribute specifier list may appear immediately before a declarator (other than the first) in a comma-separated list of declarators in a declaration of more than one identifier using a single list of specifiers and qualifiers. Such attribute specifiers apply only to the identifier before whose declarator they appear. For example, in

```
__attribute__((noreturn)) void d0 (void),
    __attribute__((format(printf, 1, 2))) d1 (const char *, ...),
     d2 (void)
```

the `noreturn` attribute applies to all the functions declared; the `format` attribute only applies to `d1`.

An attribute specifier list may appear immediately before the comma, `=` or semicolon terminating the declaration of an identifier other than a function definition. At present, such attribute specifiers apply to the declared object or function, but in future they may attach to the outermost adjacent declarator. In simple cases there is no difference, but, for example, in

```
void (****f)(void) __attribute__((noreturn));
```

at present the `noreturn` attribute applies to `f`, which causes a warning since `f` is not a function, but in future it may apply to the function `****f`. The precise semantics of what attributes in such cases will apply to are not yet specified. Where an assembler name for an object or function is specified (see Section 5.38 [Asm Labels], page 210), at present the attribute must follow the `asm` specification; in future, attributes before the `asm` specification may apply to the adjacent declarator, and those after it to the declared object or function.

An attribute specifier list may, in future, be permitted to appear after the declarator in a function definition (before any old-style parameter declarations or the function body).

Attribute specifiers may be mixed with type qualifiers appearing inside the `[]` of a parameter array declarator, in the C99 construct by which such qualifiers are applied to the pointer to which the array is implicitly converted. Such attribute specifiers apply to the pointer, not to the array, but at present this is not implemented and they are ignored.

An attribute specifier list may appear at the start of a nested declarator. At present, there are some limitations in this usage: the attributes correctly apply to the declarator, but for most individual attributes the semantics this implies are not implemented. When attribute specifiers follow the `*` of a pointer declarator, they may be mixed with any type qualifiers present. The following describes the formal semantics of this syntax. It will make the most sense if you are familiar with the formal specification of declarators in the ISO C standard.

Consider (as in C99 subclause 6.7.5 paragraph 4) a declaration `T D1`, where `T` contains declaration specifiers that specify a type *Type* (such as `int`) and `D1` is a declarator that contains an identifier *ident*. The type specified for *ident* for derived declarators whose type does not include an attribute specifier is as in the ISO C standard.

If `D1` has the form `(` *attribute-specifier-list* `D )`, and the declaration `T D` specifies the type "*derived-declarator-type-list Type*" for *ident*, then `T D1` specifies the type "*derived-declarator-type-list attribute-specifier-list Type*" for *ident*.

If `D1` has the form `*` *type-qualifier-and-attribute-specifier-list* `D`, and the declaration `T D` specifies the type "*derived-declarator-type-list Type*" for *ident*, then `T D1` specifies the type "*derived-declarator-type-list type-qualifier-and-attribute-specifier-list Type*" for *ident*.

For example,

```
void (__attribute__((noreturn)) ****f) (void);
```

specifies the type "pointer to pointer to pointer to pointer to non-returning function returning `void`". As another example,

```
char *__attribute__((aligned(8))) *f;
```

specifies the type "pointer to 8-byte-aligned pointer to `char`". Note again that this does not work with most attributes; for example, the usage of '`aligned`' and '`noreturn`' attributes given above is not yet supported.

For compatibility with existing code written for compiler versions that did not implement attributes on nested declarators, some laxity is allowed in the placing of attributes. If an attribute that only applies to types is applied to a declaration, it will be treated as applying to the type of that declaration. If an attribute that only applies to declarations is applied to the type of a declaration, it will be treated as applying to that declaration; and, for compatibility with code placing the attributes immediately before the identifier declared, such an attribute applied to a function return type will be treated as applying to the function type, and such an attribute applied to an array element type will be treated as applying to the array type. If an attribute that only applies to function types is applied to a pointer-to-function type, it will be treated as applying to the pointer target type; if such an attribute is applied to a function return type that is not a pointer-to-function type, it will be treated as applying to the function type.

5.28 Prototypes and Old-Style Function Definitions

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

```
/* Use prototypes unless the compiler is old-fashioned.  */
#ifdef __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration.  */
int isroot P((uid_t));

/* Old-style function definition.  */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

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

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

```
int isroot (uid_t);

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

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

5.29 C++ Style Comments

In GNU C, you may use C++ style comments, which start with '`//`' and continue until the end of the line. Many other C implementations allow such comments, and they are likely to be in a future C standard. However, C++ style comments are not recognized if you specify '`-ansi`', a '`-std`' option specifying a version of ISO C before C99, or '`-traditional`', since they are incompatible with traditional constructs like `dividend//*comment*/divisor`.

5.30 Dollar Signs in Identifier Names

In GNU C, you may normally use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers. However, dollar signs in identifiers are not supported on a few target machines, typically because the target assembler does not allow them.

5.31 The Character ⟨ESC⟩ in Constants

You can use the sequence '`\e`' in a string or character constant to stand for the ASCII character ⟨ESC⟩.

5.32 Inquiring on Alignment of Types or Variables

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

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

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

If the operand of `__alignof__` is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified with GCC's `__attribute__` extension (see Section 5.33 [Variable Attributes], page 184). For example, after this declaration:

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

the value of `__alignof__ (foo1.y)` is 1, even though its actual alignment is probably 2 or 4, the same as `__alignof__ (int)`.

It is an error to ask for the alignment of an incomplete type.

5.33 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Ten attributes are currently defined for variables: `aligned`, `mode`, `nocommon`, `packed`, `section`, `transparent_union`, `unused`, `deprecated`, `vector_size`, and `weak`. Some other attributes are defined for variables on particular target systems. Other attributes are available for functions (see Section 5.26 [Function Attributes], page 172) and for types (see Section 5.34 [Type Attributes], page 188). Other front ends might define more attributes (see Chapter 6 [Extensions to the C++ Language], page 249).

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

See Section 5.27 [Attribute Syntax], page 179, for details of the exact syntax for using attributes.

`aligned (`*alignment*`)`

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

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

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

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

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

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`mode (`*mode*`)`

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of '`byte`' or '`__byte__`' to indicate the mode corresponding to a one-byte integer, '`word`' or '`__word__`' for the mode of a one-word integer, and '`pointer`' or '`__pointer__`' for the mode used to represent pointers.

`nocommon` This attribute specifies requests GCC not to place a variable "common" but instead to allocate space for it directly. If you specify the '`-fno-common`' flag, GCC will do this for all variables.

Specifying the `nocommon` attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

`packed` The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

`section ("`*section-name*`")`

Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA"))) = 0;

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the `section` attribute with an *initialized* definition of a *global* variable, as shown in the example. GCC issues a warning and otherwise ignores the `section` attribute in uninitialized variable declarations.

You may only use the `section` attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the `common` (or `bss`) section and can be multiply "defined". You can force a variable to be initialized with the '`-fno-common`' flag or the `nocommon` attribute.

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

`shared` On Windows NT, in addition to putting variable definitions in a named section, the section can also be shared among all running copies of an executable or DLL. For example, this small program defines shared data by putting it in a named section `shared` and marking the section shareable:

```
int foo __attribute__((section ("shared"), shared)) = 0;

int
main()
{
  /* Read and write foo.  All running
     copies see the same value.  */
  return 0;
}
```

You may only use the `shared` attribute along with `section` attribute with a fully initialized global definition because of the way linkers work. See `section` attribute for more information.

The `shared` attribute is only available on Windows NT.

`transparent_union`
: This attribute, attached to a function parameter which is a union, means that the corresponding argument may have the type of any union member, but the argument is passed as if its type were that of the first union member. For more details see See Section 5.34 [Type Attributes], page 188. You can also use this attribute on a `typedef` for a union data type; then it applies to all function parameters with that type.

`unused`
: This attribute, attached to a variable, means that the variable is meant to be possibly unused. GCC will not produce a warning for this variable.

`deprecated`
: The `deprecated` attribute results in a warning if the variable is used anywhere in the source file. This is useful when identifying variables that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated variable, to enable users to easily find further information about why the variable is deprecated, or what they should do instead. Note that the warnings only occurs for uses:

```
extern int old_var __attribute__ ((deprecated));
extern int old_var;
int new_fn () { return old_var; }
```

results in a warning on line 3 but not line 2.

The `deprecated` attribute can also be used for functions and types (see Section 5.26 [Function Attributes], page 172, see Section 5.34 [Type Attributes], page 188.)

`vector_size (`*bytes*`)`
: This attribute specifies the vector size for the variable, measured in bytes. For example, the declaration:

```
int foo __attribute__ ((vector_size (16)));
```

causes the compiler to set the mode for `foo`, to be 16 bytes, divided into `int` sized units. Assuming a 32-bit int (a vector of 4 units of 4 bytes), the corresponding mode of `foo` will be V4SI.

This attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

Aggregates with this attribute are invalid, even if they are of the same size as a corresponding scalar. For example, the declaration:

```
struct S { int a; };
struct S  __attribute__ ((vector_size (16))) foo;
```

is invalid even if the size of the structure is the same as the size of the `int`.

`weak`
: The `weak` attribute is described in See Section 5.26 [Function Attributes], page 172.

`model (`*model-name*`)`
: Use this attribute on the M32R/D to set the addressability of an object. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

  Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction).

  Medium and large model objects may live anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses).

To specify multiple attributes, separate them by commas within the double parentheses: for example, '`__attribute__ ((aligned (16), packed))`'.

5.34 Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of `struct` and `union` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Five attributes are currently defined for types: `aligned`, `packed`, `transparent_union`, `unused`, and `deprecated`. Other attributes are defined for functions (see Section 5.26 [Function Attributes], page 172) and for variables (see Section 5.33 [Variable Attributes], page 184).

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

You may specify the `aligned` and `transparent_union` attributes either in a `typedef` declaration or just past the closing curly brace of a complete enum, struct or union type *definition* and the `packed` attribute only past the closing brace of a definition.

You may also specify attributes between the enum, struct or union tag and the name of the type rather than after the closing brace.

See Section 5.27 [Attribute Syntax], page 179, for details of the exact syntax for using attributes.

`aligned (`*alignment*`)`
: This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

  force the compiler to insure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned *at least* on a 8-byte boundary. On a Sparc, having all variables of type `struct S` aligned to 8-byte boundaries allows the compiler to use the `ldd` and `std` (doubleword load and store) instructions when copying one variable of type `struct S` to another, thus improving run-time efficiency.

  Note that the alignment of any given `struct` or `union` type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`packed` This attribute, attached to an `enum`, `struct`, or `union` type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for `struct` and `union` types is equivalent to specifying the `packed` attribute on each of the structure or union members. Specifying the '-fshort-enums' flag on the line is equivalent to specifying the packed attribute on all enum definitions.

You may only specify this attribute after a closing curly brace on an enum definition, not in a typedef declaration, unless that declaration also contains the definition of the enum.

transparent_union

This attribute, attached to a union type definition, indicates that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like const on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the wait function must accept either a value of type int * to comply with Posix, or a value of type union wait * to comply with the 4.1BSD interface. If wait's parameter were void *, wait would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, <sys/wait.h> might define the interface as follows:

```
typedef union
  {
    int *__ip;
    union wait *__up;
  } wait_status_ptr_t __attribute__ ((__transparent_union__));

pid_t wait (wait_status_ptr_t);
```

This interface allows either int * or union wait * arguments to be passed, using the int * calling convention. The program can call wait with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, wait's implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
  return waitpid (-1, p.__ip, 0);
}
```

`unused` When attached to a type (including a `union` or a `struct`), this attribute means that variables of that type are meant to appear possibly unused. GCC will not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

`deprecated`

The `deprecated` attribute results in a warning if the type is used anywhere in the source file. This is useful when identifying types that are expected to be removed in a future version of a program. If possible, the warning also includes the location of the declaration of the deprecated type, to enable users to easily find further information about why the type is deprecated, or what they should do instead. Note that the warnings only occur for uses and then only if the type is being applied to an identifier that itself is not being declared as deprecated.

```
typedef int T1 __attribute__ ((deprecated));
T1 x;
typedef T1 T2;
T2 y;
typedef T1 T3 __attribute__ ((deprecated));
T3 z __attribute__ ((deprecated));
```

results in a warning on line 2 and 3 but not lines 4, 5, or 6. No warning is issued for line 4 because T2 is not explicitly deprecated. Line 5 has no warning because T3 is explicitly deprecated. Similarly for line 6.

The `deprecated` attribute can also be used for functions and variables (see Section 5.26 [Function Attributes], page 172, see Section 5.33 [Variable Attributes], page 184.)

To specify multiple attributes, separate them by commas within the double parentheses: for example, '`__attribute__ ((aligned (16), packed))`'.

5.35 An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GCC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use '`-O`', no function is really inline.

Inline functions are included in the ISO C99 standard, but there are currently substantial differences between what GCC implements and what the ISO C99 standard requires.

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 ISO C programs, write `__inline__` instead of `inline`. See Section 5.40 [Alternate Keywords], page 213.) You can also make all "simple enough" functions inline with the option '`-finline-functions`'.

Note that certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of varargs, use of alloca, use of variable sized data types (see Section 5.14 [Variable Length], page 165), use of computed goto (see Section 5.3 [Labels as Values], page 157), use of nonlocal goto, and nested functions (see Section 5.4 [Nested Functions], page 158). Using '`-Winline`' will warn when a function marked `inline` could not be substituted, and will give the reason for the failure.

Note that in C and Objective-C, unlike C++, the `inline` keyword does not affect the linkage of the function.

GCC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared `inline`. (You can override this with '`-fno-default-inline`'; see Section 3.5 [Options Controlling C++ Dialect], page 24.)

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, GCC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

For future compatibility with when GCC implements ISO C99 semantics for inline functions, it is best to use `static inline` only. (The existing semantics will remain available when '`-std=gnu89`' is specified, but eventually the default will be '`-std=gnu99`' and that will implement the C99 semantics, though it does not do so yet.)

GCC does not inline any functions when not optimizing unless you specify the '`always_inline`' attribute for the function, like this:

```
/* Prototype.  */
inline void foo (const char) __attribute__((always_inline));
```

5.36 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can specify the operands of the instruction using C expressions. This means you need not guess which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has '`"f"`' as its operand constraint, saying that a floating point register is required. The '`=`' in '`=f`' indicates that the operand is an output; all output operands' constraints must use '`=`'. The constraints use the same language used in the machine description (see Section 5.37 [Constraints], page 198).

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 the operands within each group. The total number of operands is currently limited to 30; this limitation may be lifted in some future version of GCC.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go.

As of GCC version 3.1, it is also possible to specify input and output operands using symbolic names which can be referenced within the assembler code. These names are specified inside square brackets preceding the constraint string, and can be referenced inside the assembler code using `%[`*name*`]` instead of a percentage sign followed by the operand number. Using named operands the above example could look like:

```
asm ("fsinx %[angle],%[output]"
     : [output] "=f" (result)
     : [angle] "f" (angle));
```

Note that the symbolic operand names have no relation whatsoever to other C identifiers. You may use any name you like, even those of existing C symbols, but must ensure that no two operands within the same assembler construct use the same symbolic name.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means or even whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit-field), your constraint must allow a register. In that case, GCC will use the register as the output of the `asm`, and then store that register into the output.

The ordinary output operands must be write-only; GCC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm supports input-output or read-write operands. Use the constraint character '+' to indicate such an operand and list it with the output operands.

When the constraints for the read-write operand (or the operand in which only some of the bits are to be changed) allows a register, you may, as an alternative, 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 number in constraint is allowed only in an input operand and it must refer to an output operand.

Only a number in the constraint can guarantee that one operand will be in the same place as another. The mere fact that foo is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work reliably:

```
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; GCC knows no reason not to do so. For example, the compiler might find a copy of the value of foo in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to foo's own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GCC can't tell that.

As of GCC version 3.1, one may write [*name*] instead of the operand number for a matching constraint. For example:

```
asm ("cmoveq %1,%2,%[result]"
     : [result] "=r"(result)
     : "r" (test), "r"(new), "[result]"(old));
```

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the VAX:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

You may not write a clobber description in a way that overlaps with an input or output operand. For example, you may not have an operand describing a register class with one member if you mention that register in the clobber list. There is no way for you to specify that an input operand is modified without also specifying it as an output operand. Note that if all the output operands you specify are for this purpose (and hence unused), you will then also need to specify volatile for the asm construct, as described below, to prevent GCC from deleting the asm statement as unused.

If you refer to a particular hardware register from the assembler code, you will probably have to list the register after the third colon to tell the compiler the register's value is modified. In some assemblers, the register names begin with '%'; to produce one '%' in the assembler code, you must write '%%' in the input.

If your assembler instruction can alter the condition code register, add 'cc' to the list of clobbered registers. GCC on some machines represents the condition codes as a specific hardware register; 'cc' serves to name this register. On other machines, the condition code is handled differently, and specifying 'cc' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add 'memory' to the list of clobbered registers. This will cause GCC to not keep memory values cached in registers across the assembler instruction. You will also want to add the `volatile` keyword if the memory affected is not listed in the inputs or outputs of the `asm`, as the 'memory' clobber does not count as a side-effect of the `asm`.

You can put multiple assembler instructions together in a single `asm` template, separated by the characters normally used in assembly code for the system. A combination that works in most places is a newline to break the line, plus a tab character to move to the instruction field (written as '\n\t'). Sometimes semicolons can be used, if the assembler allows semicolons as a line-breaking character. Note that some assembler dialects use semicolons to start a comment. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes the subroutine `_foo` accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9\n\tmovl %1,r10\n\tcall _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '&' constraint modifier, GCC may allocate it in the same register as an unrelated input operand, on the assumption 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. See Section 5.37.3 [Modifiers], page 201.

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\n\tfrob %1\n\tbeq 0f\n\tmov #1,%0\n0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one `asm` to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
```

```
        __value; })
```

Here the variable `__arg` is used to make sure that the instruction operates on a proper `double` value, and to accept only those arguments `x` which can convert automatically to a `double`.

Another way to make sure the instruction operates on the correct data type is to use a cast in the `asm`. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would warn about using a pointer unless the caller explicitly casts it.

If an `asm` has output operands, GCC assumes for optimization purposes the instruction has no side effects except to change the output operands. This does not mean 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 get_and_set_priority(new)              \
({ int __old;                                  \
   asm volatile ("get_and_set_priority %0, %1" \
                 : "=g" (__old) : "g" (new));  \
   __old; })
```

If you write an `asm` instruction with no outputs, GCC will know the instruction has side-effects and will not delete the instruction or move it outside of loops.

The `volatile` keyword indicates that the instruction has important side-effects. GCC will not delete a volatile `asm` if it is reachable. (The instruction can still be deleted if GCC can prove that control-flow will never reach the location of the instruction.) In addition, GCC will not reschedule instructions across a volatile `asm` instruction. For example:

```
*(volatile int *)addr = foo;
asm volatile ("eieio" : : );
```

Assume `addr` contains the address of a memory mapped device register. The PowerPC `eieio` instruction (Enforce In-order Execution of I/O) tells the CPU to make sure that the store to that device register happens before it issues any other I/O.

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`. Also, GCC will perform some optimizations across a volatile `asm` instruction; GCC does not "forget everything" when it encounters a volatile `asm` instruction the way some other compilers do.

An `asm` instruction without any operands or clobbers (an "old style" `asm`) will be treated identically to a volatile `asm` instruction.

It is 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 "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

For reasons similar to those described above, it is not possible to give an assembler instruction access to the condition code left by previous instructions.

If you are writing a header file that should be includable in ISO C programs, write `__asm__` instead of `asm`. See Section 5.40 [Alternate Keywords], page 213.

5.36.1 i386 floating point asm operands

There are several rules on the usage of stack-like regs in asm_operands insns. These rules apply only to the operands that are stack-like regs:

1. Given a set of input regs that die in an asm_operands, it is necessary to know which are implicitly popped by the asm, and which must be explicitly popped by gcc.

   An input reg that is implicitly popped by the asm must be explicitly clobbered, unless it is constrained to match an output operand.
2. For any input reg that is implicitly popped by an asm, it is necessary to know how to adjust the stack to compensate for the pop. If any non-popped input is closer to the top of the reg-stack than the implicitly popped reg, it would not be possible to know what the stack looked like—it's not clear how the rest of the stack "slides up".

   All implicitly popped input regs must be closer to the top of the reg-stack than any input that is not implicitly popped.

   It is possible that if an input dies in an insn, reload might use the input reg for an output reload. Consider this example:

   ```
   asm ("foo" : "=t" (a) : "f" (b));
   ```

   This asm says that input B is not popped by the asm, and that the asm pushes a result onto the reg-stack, i.e., the stack is one deeper after the asm than it was before. But, it is possible that reload will think that it can use the same reg for both the input and the output, if input B dies in this insn.

   If any input operand uses the `f` constraint, all output reg constraints must use the `&` earlyclobber.

   The asm above would be written as

   ```
   asm ("foo" : "=&t" (a) : "f" (b));
   ```
3. Some operands need to be in particular places on the stack. All output operands fall in this category—there is no other way to know which regs the outputs appear in unless the user indicates this in the constraints.

   Output operands must specifically indicate which reg an output appears in after an asm. `=f` is not allowed: the operand constraints must select a class with a single reg.
4. Output operands may not be "inserted" between existing stack regs. Since no 387 opcode uses a read/write operand, all output operands are dead before the asm_operands, and are pushed by the asm_operands. It makes no sense to push anywhere but the top of the reg-stack.

Output operands must start at the top of the reg-stack: output operands may not "skip" a reg.

5. Some asm statements may need extra stack space for internal calculations. This can be guaranteed by clobbering stack registers unrelated to the inputs and outputs.

Here are a couple of reasonable asms to want to write. This asm takes one input, which is internally popped, and produces two outputs.

```
asm ("fsincos" : "=t" (cos), "=u" (sin) : "0" (inp));
```

This asm takes two inputs, which are popped by the `fyl2xp1` opcode, and replaces them with one output. The user must code the `st(1)` clobber for reg-stack.c to know that `fyl2xp1` pops both inputs.

```
asm ("fyl2xp1" : "=t" (result) : "0" (x), "u" (y) : "st(1)");
```

5.37 Constraints for `asm` Operands

Here are specific details on what constraint letters you can use with `asm` operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

5.37.1 Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

whitespace

Whitespace characters are ignored and can be inserted at any position except the first. This enables each alternative for different operands to be visually aligned in the machine description even if they have different number of constraints and modifiers.

'`m`'

A memory operand is allowed, with any kind of address that the machine supports in general.

'`o`'

A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter '`o`' is valid only when accompanied by both '`<`' (if the target machine has predecrement addressing) and '`>`' (if the target machine has preincrement addressing).

`V`
: A memory operand that is not offsettable. In other words, anything that would fit the `m` constraint but not the `o` constraint.

`<`
: A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.

`>`
: A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

`r`
: A register operand is allowed provided that it is in a general register.

`i`
: An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

`n`
: An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use `n` rather than `i`.

`I`, `J`, `K`, . . . `P`
: Other letters in the range `I` through `P` may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, `I` is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

`E`
: An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

`F`
: An immediate floating operand (expression code `const_double`) is allowed.

`G`, `H`
: `G` and `H` may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

`s`
: An immediate integer operand whose value is not an explicit integer is allowed.

  This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use `s` instead of `i`? Sometimes it allows better code to be generated.

  For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a `moveq` instruction. We arrange for this to happen by defining the letter `K` to mean "any integer outside the range -128 to 127", and then specifying `Ks` in the operand constraints.

`g`
: Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

`X`
: Any operand whatsoever is allowed.

`0`, `1`, `2`, . . . `9`
: An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This number is allowed to be more than a single digit. If multiple digits are encountered consecutavely, they are interpreted as a single decimal integer. There is scant chance for ambiguity, since to-date it has never been desirable that '`10`' be interpreted as matching either operand 1 *or* operand 0. Should this be desired, one can use multiple alternatives instead.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which `asm` distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

'`p`' An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

'`p`' in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

other-letters

Other letters can be defined in machine-dependent fashion to stand for particular classes of registers or other arbitrary operand types. '`d`', '`a`' and '`f`' are defined on the 68000/68020 to stand for data, address and floating point registers.

5.37.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the '`?`' and '`!`' characters:

`?` Disparage slightly the alternative that the '`?`' appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each '`?`' that appears in it.

! Disparage severely the alternative that the '!' appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

5.37.3 Constraint Modifier Characters

Here are constraint modifier characters.

'=' Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

'+' Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. '=' identifies an output; '+' identifies an operand that is both input and output; all other operands are assumed to be input only.

If you specify '=' or '+' in a constraint, you put it in the first character of the constraint string.

'&' Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

'&' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires '&' while others do not. See, for example, the 'movdf' insn of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the 'mulsi3' insn of the ARM.

'&' does not obviate the need to write '='.

'%' Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints.

'#' Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

'*' Says that the following character should be ignored when choosing register preferences. '*' has no effect on the meaning of the constraint as a constraint, and no effect on reloading.

5.37.4 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures.

The most commonly used constraints are '`m`' and '`r`' (for memory and general-purpose registers respectively; see Section 5.37.1 [Simple Constraints], page 198), and '`I`', usually the letter indicating the most common immediate-constant format.

For each machine architecture, the '`config/`*machine*`/`*machine*`.h`' file defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly interesting for `asm`. The constraints are defined through these macros:

`REG_CLASS_FROM_LETTER`
: Register class constraints (usually lower case).

`CONST_OK_FOR_LETTER_P`
: Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually upper case).

`CONST_DOUBLE_OK_FOR_LETTER_P`
: Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually upper case).

`EXTRA_CONSTRAINT`
: Special cases of registers or memory. This macro is not required, and is only defined for some machines.

Inspecting these macro definitions in the compiler source for your machine is the best way to be certain you have the right constraints. However, here is a summary of the machine-dependent constraints available on some particular machines.

ARM family—'`arm.h`'

`f` Floating-point register

`F` One of the floating-point constants 0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0 or 10.0

`G` Floating-point constant that would satisfy the constraint '`F`' if it were negated

`I` Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2

`J` Integer in the range −4095 to 4095

`K` Integer that satisfies constraint '`I`' when inverted (ones complement)

`L` Integer that satisfies constraint '`I`' when negated (twos complement)

`M` Integer in the range 0 to 32

`Q` A memory reference where the exact address is in a single register ("`m`" is preferable for `asm` statements)

`R` An item in the constant pool

`S` A symbol in the text segment of the current file

AMD 29000 family—'`a29k.h`'

`l` Local register 0

`b` Byte Pointer ('`BP`') register

`q` '`Q`' register

`h` Special purpose register

`A` First accumulator register

`a` Other accumulator register

`f` Floating point register

`I` Constant greater than 0, less than 0x100

`J` Constant greater than 0, less than 0x10000

`K` Constant whose high 24 bits are on (1)

`L` 16-bit constant whose high 8 bits are on (1)

`M` 32-bit constant whose high 16 bits are on (1)

`N` 32-bit negative constant that fits in 8 bits

`O` The constant 0x80000000 or, on the 29050, any 32-bit constant whose low 16 bits are 0.

`P` 16-bit negative constant that fits in 8 bits

`G`
`H` A floating point constant (in `asm` statements, use the machine independent '`E`' or '`F`' instead)

AVR family—'`avr.h`'

`l` Registers from r0 to r15

`a` Registers from r16 to r23

`d` Registers from r16 to r31

`w` Registers from r24 to r31. These registers can be used in '`adiw`' command

`e` Pointer register (r26–r31)

`b` Base pointer register (r28–r31)

`q` Stack pointer register (SPH:SPL)

`t` Temporary register r0

`x` Register pair X (r27:r26)

`y` Register pair Y (r29:r28)

`z` Register pair Z (r31:r30)

`I` Constant greater than −1, less than 64

`J` Constant greater than −64, less than 1

`K` Constant integer 2

`L` Constant integer 0

`M` Constant that fits in 8 bits

`N` Constant integer −1

`O` Constant integer 8, 16, or 24

`P` Constant integer 1

`G` A floating point constant 0.0

IBM RS6000—'`rs6000.h`'

`b` Address base register

`f` Floating point register

`h` '`MQ`', '`CTR`', or '`LINK`' register

`q` '`MQ`' register

`c` '`CTR`' register

`l` '`LINK`' register

`x` '`CR`' register (condition register) number 0

`y` '`CR`' register (condition register)

`z` '`FPMEM`' stack memory for FPR-GPR transfers

`I` Signed 16-bit constant

`J` Unsigned 16-bit constant shifted left 16 bits (use '`L`' instead for `SImode` constants)

`K` Unsigned 16-bit constant

`L` Signed 16-bit constant shifted left 16 bits

`M` Constant larger than 31

`N` Exact power of 2

`O` Zero

`P` Constant whose negation is a signed 16-bit constant

`G` Floating point constant that can be loaded into a register with one instruction per word

`Q` Memory operand that is an offset from a register ('`m`' is preferable for `asm` statements)

`R` AIX TOC entry

`S` Constant suitable as a 64-bit mask operand

`T` Constant suitable as a 32-bit mask operand

`U` System V Release 4 small data area reference

Intel 386—'`i386.h`'

`q` 'a', `b`, `c`, or `d` register for the i386. For x86-64 it is equivalent to '`r`' class. (for 8-bit instructions that do not use upper halves)

`Q` 'a', `b`, `c`, or `d` register. (for 8-bit instructions, that do use upper halves)

`R` Legacy register—equivalent to `r` class in i386 mode. (for non-8-bit registers used together with 8-bit upper halves in a single instruction)

`A` Specifies the '`a`' or '`d`' registers. This is primarily useful for 64-bit integer values (when in 32-bit mode) intended to be returned with the '`d`' register holding the most significant bits and the '`a`' register holding the least significant bits.

`f` Floating point register

`t` First (top of stack) floating point register

`u` Second floating point register

`a` '`a`' register

`b` '`b`' register

`c` '`c`' register

`d` '`d`' register

`D` '`di`' register

`S` '`si`' register

`x` '`xmm`' SSE register

`y` MMX register

`I` Constant in range 0 to 31 (for 32-bit shifts)

`J` Constant in range 0 to 63 (for 64-bit shifts)

`K` '`0xff`'

`L` '`0xffff`'

`M` 0, 1, 2, or 3 (shifts for `lea` instruction)

`N` Constant in range 0 to 255 (for `out` instruction)

`Z` Constant in range 0 to `0xffffffff` or symbolic reference known to fit specified range. (for using immediates in zero extending 32-bit to 64-bit x86-64 instructions)

`e` Constant in range −2147483648 to 2147483647 or symbolic reference known to fit specified range. (for using immediates in 64-bit x86-64 instructions)

`G` Standard 80387 floating point constant

Intel 960—'i960.h'

`f` Floating point register (`fp0` to `fp3`)

`l` Local register (`r0` to `r15`)

`b` Global register (`g0` to `g15`)

`d` Any local or global register

`I` Integers from 0 to 31

`J` 0

`K` Integers from −31 to 0

`G` Floating point 0

`H` Floating point 1

MIPS—'mips.h'

`d` General-purpose integer register

`f` Floating-point register (if available)

`h` '`Hi`' register

`l` '`Lo`' register

`x` '`Hi`' or '`Lo`' register

`y` General-purpose integer register

`z` Floating-point status register

`I` Signed 16-bit constant (for arithmetic instructions)

`J` Zero

`K` Zero-extended 16-bit constant (for logic instructions)

`L` Constant with low 16 bits zero (can be loaded with `lui`)

`M` 32-bit constant which requires two instructions to load (a constant which is not '`I`', '`K`', or '`L`')

`N` Negative 16-bit constant

`O` Exact power of two

`P` Positive 16-bit constant

`G` Floating point zero

`Q` Memory reference that can be loaded with more than one instruction ('`m`' is preferable for `asm` statements)

`R` Memory reference that can be loaded with one instruction ('`m`' is preferable for `asm` statements)

`S` Memory reference in external OSF/rose PIC format ('`m`' is preferable for `asm` statements)

Motorola 680x0—'`m68k.h`'

`a` Address register

`d` Data register

`f` 68881 floating-point register, if available

`x` Sun FPA (floating-point) register, if available

`y` First 16 Sun FPA registers, if available

`I` Integer in the range 1 to 8

`J` 16-bit signed number

`K` Signed number whose magnitude is greater than 0x80

`L` Integer in the range −8 to −1

`M` Signed number whose magnitude is greater than 0x100

`G` Floating point constant that is not a 68881 constant

`H` Floating point constant that can be used by Sun FPA

Motorola 68HC11 & 68HC12 families—'`m68hc11.h`'

`a` Register 'a'

`b` Register 'b'

`d` Register 'd'

`q` An 8-bit register

`t` Temporary soft register _.tmp

`u` A soft register _.d1 to _.d31

`w` Stack pointer register

`x` Register 'x'

`y` Register 'y'

`z` Pseudo register 'z' (replaced by 'x' or 'y' at the end)

`A` An address register: x, y or z

`B` An address register: x or y

`D` Register pair (x:d) to form a 32-bit value

`L` Constants in the range −65536 to 65535

`M` Constants whose 16-bit low part is zero

`N` Constant integer 1 or −1

`O` Constant integer 16

`P` Constants in the range −8 to 2

SPARC—'`sparc.h`'

`f` Floating-point register that can hold 32- or 64-bit values.

`e` Floating-point register that can hold 64- or 128-bit values.

`I` Signed 13-bit constant

`J` Zero

`K` 32-bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)

`L` A constant in the range supported by `movcc` instructions

`M` A constant in the range supported by `movrcc` instructions

`N` Same as '`K`', except that it verifies that bits that are not in the lower 32-bit range are all zero. Must be used instead of '`K`' for modes wider than `SImode`

`G` Floating-point zero

`H` Signed 13-bit constant, sign-extended to 32 or 64 bits

`Q` Floating-point constant whose integral representation can be moved into an integer register using a single sethi instruction

`R` Floating-point constant whose integral representation can be moved into an integer register using a single mov instruction

`S` Floating-point constant whose integral representation can be moved into an integer register using a high/lo_sum instruction sequence

`T` Memory address aligned to an 8-byte boundary

`U` Even register

`W` Memory address for '`e`' constraint registers.

TMS320C3x/C4x—'`c4x.h`'

`a` Auxiliary (address) register (ar0-ar7)

`b` Stack pointer register (sp)

`c` Standard (32-bit) precision integer register

`f` Extended (40-bit) precision register (r0-r11)

`k` Block count register (bk)

`q` Extended (40-bit) precision low register (r0-r7)

`t` Extended (40-bit) precision register (r0-r1)

`u` Extended (40-bit) precision register (r2-r3)

`v` Repeat count register (rc)

`x` Index register (ir0-ir1)

`y` Status (condition code) register (st)

`z` Data page register (dp)

`G` Floating-point zero

`H` Immediate 16-bit floating-point constant

`I` Signed 16-bit constant

`J` Signed 8-bit constant

`K` Signed 5-bit constant

`L` Unsigned 16-bit constant

`M` Unsigned 8-bit constant

`N` Ones complement of unsigned 16-bit constant

`O` High 16-bit constant (32-bit constant with 16 LSBs zero)

`Q` Indirect memory reference with signed 8-bit or index register displacement

`R` Indirect memory reference with unsigned 5-bit displacement

`S` Indirect memory reference with 1 bit or index register displacement

`T` Direct memory reference

`U` Symbolic address

S/390 and zSeries—‘`s390.h`’

`a` Address register (general purpose register except r0)

`d` Data register (arbitrary general purpose register)

`f` Floating-point register

`I` Unsigned 8-bit constant (0–255)

`J` Unsigned 12-bit constant (0–4095)

`K` Signed 16-bit constant (−32768–32767)

`L` Unsigned 16-bit constant (0–65535)

`Q` Memory reference without index register

`S` Symbolic constant suitable for use with the `larl` instruction

Xstormy16—‘`stormy16.h`’

`a` Register r0.

`b` Register r1.

`c` Register r2.

`d` Register r8.

`e` Registers r0 through r7.

`t` Registers r0 and r1.

`y` The carry register.

`z` Registers r8 and r9.

`I` A constant between 0 and 3 inclusive.

`J` A constant that has exactly one bit set.

`K` A constant that has exactly one bit clear.

`L` A constant between 0 and 255 inclusive.

`M` A constant between −255 and 0 inclusive.

`N` A constant between −3 and 0 inclusive.

`O` A constant between 1 and 4 inclusive.

`P` A constant between −4 and −1 inclusive.

`Q` A memory reference that is a stack push.

`R` A memory reference that is a stack pop.

`S` A memory reference that refers to an constant address of known value.

`T` The register indicated by Rx (not implemented yet).

`U` A constant that is not between 2 and 15 inclusive.

Xtensa—'`xtensa.h`'

`a` General-purpose 32-bit register

`b` One-bit boolean register

`A` MAC16 40-bit accumulator register

`I` Signed 12-bit integer constant, for use in MOVI instructions

`J` Signed 8-bit integer constant, for use in ADDI instructions

`K` Integer constant valid for BccI instructions

`L` Unsigned constant valid for BccUI instructions

5.38 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

It does not make sense to use this feature with a non-static local variable since such variables do not have assembler names. If you are trying to put the variable in a particular register, see Section 5.39 [Explicit Reg Vars], page 211. GCC presently accepts such code with a warning, but will probably be changed to issue an error, rather than a warning, in the future.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");

func (x, y)
     int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GCC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

5.39 Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses. Stores into local register variables may be deleted when they appear to be dead according to dataflow analysis. References to local register variables may be deleted or moved or simplified.

  These local variables are sometimes convenient for use with the extended `asm` feature (see Section 5.36 [Extended Asm], page 193), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

5.39.1 Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register `a5` would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected magically by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e. in a different source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. (If you are prepared to recompile `qsort` with the same global register variable, you can solve this problem.)

If you want to recompile `qsort` or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option '`-ffixed-`*reg*'. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` will not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing will happen regardless of what `longjmp` does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as `getwd`, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

5.39.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("a5");
```

Here a5 is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see Section 5.36 [Extended Asm], page 193). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register %a5.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. However, these registers are made unavailable for use in the reload pass; excessive use of this feature leaves the compiler too few available registers to compile certain functions.

This option does not guarantee that GCC will generate code that has this variable in the register you specify at all times. You may not code an explicit reference to this register in an asm statement and assume it will always refer to this variable.

Stores into local register variables may be deleted when they appear to be dead according to dataflow analysis. References to local register variables may be deleted or moved or simplified.

5.40 Alternate Keywords

The option '-traditional' disables certain keywords; '-ansi' and the various '-std' options disable certain others. This causes trouble when you want to use GNU C extensions, or ISO C features, in a general-purpose header file that should be usable by all programs, including ISO C programs and traditional ones. The keywords asm, typeof and inline cannot be used since they won't work in a program compiled with '-ansi' (although inline can be used in a program compiled with '-std=c99'), while the keywords const, volatile, signed, typeof and inline won't work in a program compiled with '-traditional'. The ISO C99 keyword restrict is only available when '-std=gnu99' (which will eventually be the default) or '-std=c99' (or the equivalent '-std=iso9899:1999') is used.

The way to solve these problems is to put '__' at the beginning and end of each problematical keyword. For example, use __asm__ instead of asm, __const__ instead of const, and __inline__ instead of inline.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'-pedantic' and other options cause warnings for many GNU C extensions. You can prevent such warnings within one expression by writing __extension__ before the expression. __extension__ has no effect aside from this.

5.41 Incomplete `enum` Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

This extension is not supported by GNU C++.

5.42 Function Names as Strings

GCC predefines two magic identifiers to hold the name of the current function. The identifier `__FUNCTION__` holds the name of the function as it appears in the source. The identifier `__PRETTY_FUNCTION__` holds the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

```
extern "C" {
extern int printf (char *, ...);
}

class a {
 public:
  sub (int i)
    {
      printf ("__FUNCTION__ = %s\n", __FUNCTION__);
      printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
    }
};

int
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = int  a::sub (int)
```

The compiler automagically replaces the identifiers with a string literal containing the appropriate name. Thus, they are neither preprocessor macros, like `__FILE__` and `__LINE__`, nor variables. This means that they catenate with other string literals, and that they can be used to initialize char arrays. For example

```
char here[] = "Function " __FUNCTION__ " in " __FILE__;
```

On the other hand, '`#ifdef __FUNCTION__`' does not have any special meaning inside a function, since the preprocessor does not do anything special with the identifier `__FUNCTION__`.

Note that these semantics are deprecated, and that GCC 3.2 will handle `__FUNCTION__` and `__PRETTY_FUNCTION__` the same way as `__func__`. `__func__` is defined by the ISO standard C99:

> The identifier `__func__` is implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> appeared, where function-name is the name of the lexically-enclosing function. This name is the unadorned name of the function.

By this definition, `__func__` is a variable, not a string literal. In particular, `__func__` does not catenate with other string literals.

In `C++`, `__FUNCTION__` and `__PRETTY_FUNCTION__` are variables, declared in the same way as `__func__`.

5.43 Getting the Return or Frame Address of a Function

These functions may be used to get information about the callers of a function.

`void * __builtin_return_address (unsigned int` *level*`)` Built-in Function

This function returns the return address of the current function, or of one of its callers. The *level* argument is number of frames to scan up the call stack. A value of `0` yields the return address of the current function, a value of `1` yields the return address of the caller of the current function, and so forth.

The *level* argument must be a constant integer.

On some machines it may be impossible to determine the return address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function will return `0` or a random value. In addition, `__builtin_frame_address` may be used to determine if the top of the stack has been reached.

This function should only be used with a nonzero argument for debugging purposes.

`void * __builtin_frame_address (unsigned int` *level*`)` Built-in Function

This function is similar to `__builtin_return_address`, but it returns the address of the function frame rather than the return address of the function. Calling `__builtin_frame_address` with a value of `0` yields the frame address of the current function, a value of `1` yields the frame address of the caller of the current function, and so forth.

The frame is the area on the stack which holds local variables and saved registers. The frame address is normally the address of the first word pushed on to the stack by the function. However, the exact definition depends upon the processor and the calling convention. If the processor has a dedicated frame pointer register, and the function has a frame, then `__builtin_frame_address` will return the value of the frame pointer register.

On some machines it may be impossible to determine the frame address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function will return `0` if the first frame pointer is properly initialized by the startup code.

This function should only be used with a nonzero argument for debugging purposes.

5.44 Using vector instructions through built-in functions

On some targets, the instruction set contains SIMD vector instructions that operate on multiple values contained in one large register at the same time. For example, on the i386 the MMX, 3Dnow! and SSE extensions can be used this way.

The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate `typedef`:

```
typedef int v4si __attribute__ ((mode(V4SI)));
```

The base type `int` is effectively ignored by the compiler, the actual properties of the new type `v4si` are defined by the `__attribute__`. It defines the machine mode to be used; for vector types these have the form `V`*nB*; *n* should be the number of elements in the vector, and *B* should be the base mode of the individual elements. The following can be used as base modes:

`QI` An integer that is as wide as the smallest addressable unit, usually 8 bits.

`HI` An integer, twice as wide as a QI mode integer, usually 16 bits.

`SI` An integer, four times as wide as a QI mode integer, usually 32 bits.

`DI` An integer, eight times as wide as a QI mode integer, usually 64 bits.

`SF` A floating point value, as wide as a SI mode integer, usually 32 bits.

`DF` A floating point value, as wide as a DI mode integer, usually 64 bits.

Not all base types or combinations are always valid; which modes can be used is determined by the target machine. For example, if targetting the i386 MMX extensions, only `V8QI`, `V4HI` and `V2SI` are allowed modes.

There are no `V1xx` vector modes - they would be identical to the corresponding base mode.

There is no distinction between signed and unsigned vector modes. This distinction is made by the operations that perform on the vectors, not by the data type.

The types defined in this manner are somewhat special, they cannot be used with most normal C operations (i.e., a vector addition can *not* be represented by a normal addition of two vector type variables). You can declare only variables and use them in function calls and returns, as well as in assignments and some casts. It is possible to cast from one vector type to another, provided they are of the same size (in fact, you can also cast vectors to and from other datatypes of the same size).

A port that supports vector operations provides a set of built-in functions that can be used to operate on vectors. For example, a function to add two vectors and multiply the result by a third could look like this:

```
v4si f (v4si a, v4si b, v4si c)
{
  v4si tmp = __builtin_addv4si (a, b);
  return __builtin_mulv4si (tmp, c);
}
```

5.45 Other built-in functions provided by GCC

GCC provides a large number of built-in functions other than the ones mentioned above. Some of these are for internal use in the processing of exceptions or variable-length argument lists and will not be documented here because they may change from time to time; we do not recommend general use of these functions.

The remaining functions are provided for optimization purposes.

GCC includes built-in versions of many of the functions in the standard C library. The versions prefixed with `__builtin_` will always be treated as having the same meaning as the C library function even if you specify the '`-fno-builtin`' option. (see Section 3.4 [C Dialect Options], page 19) Many of these functions are only optimized in certain cases; if they are not optimized in a particular case, a call to the library function will be emitted.

The functions `abort`, `exit`, `_Exit` and `_exit` are recognized and presumed not to return, but otherwise are not built in. `_exit` is not recognized in strict ISO C mode ('`-ansi`', '`-std=c89`' or '`-std=c99`'). `_Exit` is not recognized in strict C89 mode ('`-ansi`' or '`-std=c89`').

Outside strict ISO C mode, the functions `alloca`, `bcmp`, `bzero`, `index`, `rindex`, `ffs`, `fputs_unlocked`, `printf_unlocked` and `fprintf_unlocked` may be handled as built-in functions. All these functions have corresponding versions prefixed with `__builtin_`, which may be used even in strict C89 mode.

The ISO C99 functions `conj`, `conjf`, `conjl`, `creal`, `crealf`, `creall`, `cimag`, `cimagf`, `cimagl`, `llabs` and `imaxabs` are handled as built-in functions except in strict ISO C89 mode. There are also built-in versions of the ISO C99 functions `cosf`, `cosl`, `fabsf`, `fabsl`, `sinf`, `sinl`, `sqrtf`, and `sqrtl`, that are recognized in any mode since ISO C89 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with `__builtin_`.

The ISO C89 functions `abs`, `cos`, `fabs`, `fprintf`, `fputs`, `labs`, `memcmp`, `memcpy`, `memset`, `printf`, `sin`, `sqrt`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, and `strstr` are all recognized as built-in functions unless '`-fno-builtin`' is specified (or '`-fno-builtin-`*function*' is specified for an individual function). All of these functions have corresponding versions prefixed with `__builtin_`.

GCC provides built-in versions of the ISO C99 floating point comparison macros that avoid raising exceptions for unordered operands. They have the same names as the standard macros (`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered`) , with `__builtin_` prefixed. We intend for a library implementor to be able to simply `#define` each standard macro to its built-in equivalent.

int `__builtin_types_compatible_p` (*type1*, *type2*) Built-in Function

You can use the built-in function `__builtin_types_compatible_p` to determine whether two types are the same.

This built-in function returns 1 if the unqualified versions of the types *type1* and *type2* (which are types, not expressions) are compatible, 0 otherwise. The result of this built-in function can be used in integer constant expressions.

This built-in function ignores top level qualifiers (e.g., `const`, `volatile`). For example, `int` is equivalent to `const int`.

The type `int[]` and `int[5]` are compatible. On the other hand, `int` and `char *` are not compatible, even if the size of their types, on the particular architecture are the same. Also, the amount of pointer indirection is taken into account when determining similarity. Consequently, `short *` is not similar to `short **`. Furthermore, two types that are typedefed are considered compatible if their underlying types are compatible.

An `enum` type is considered to be compatible with another `enum` type. For example, `enum {foo, bar}` is similar to `enum {hot, dog}`.

You would typically use this function in code whose execution varies depending on the arguments' types. For example:

```
#define foo(x)                                                  \
  ({                                                           \
    typeof (x) tmp;                                             \
    if (__builtin_types_compatible_p (typeof (x), long double)) \
      tmp = foo_long_double (tmp);                              \
    else if (__builtin_types_compatible_p (typeof (x), double)) \
      tmp = foo_double (tmp);                                   \
    else if (__builtin_types_compatible_p (typeof (x), float))  \
      tmp = foo_float (tmp);                                    \
    else                                                        \
      abort ();                                                 \
    tmp;                                                        \
  })
```

Note: This construct is only available for C.

type `__builtin_choose_expr` (*const_exp*, *exp1*, *exp2*) Built-in Function

You can use the built-in function `__builtin_choose_expr` to evaluate code depending on the value of a constant expression. This built-in function returns *exp1* if *const_exp*, which is a constant expression that must be able to be determined at compile time, is nonzero. Otherwise it returns 0.

This built-in function is analogous to the '`? :`' operator in C, except that the expression returned has its type unaltered by promotion rules. Also, the built-in function does not evaluate the expression that was not chosen. For example, if *const_exp* evaluates to true, *exp2* is not evaluated even if it has side-effects.

This built-in function can return an lvalue if the chosen argument is an lvalue.

If *exp1* is returned, the return type is the same as *exp1*'s type. Similarly, if *exp2* is returned, its return type is the same as *exp2*.

Example:

```
#define foo(x)                                                          \
  __builtin_choose_expr (__builtin_types_compatible_p (typeof (x), double),  \
    foo_double (x),                                                     \
    __builtin_choose_expr (__builtin_types_compatible_p (typeof (x), float), \
      foo_float (x),                                                    \
      /* The void expression results in a compile-time error            \
         when assigning the result to something.  */                    \
      (void)0))
```

Note: This construct is only available for C. Furthermore, the unused expression (*exp1* or *exp2* depending on the value of *const_exp*) may still generate syntax errors. This may change in future revisions.

`int __builtin_constant_p (`*exp*`)` Built-in Function

You can use the built-in function `__builtin_constant_p` to determine if a value is known to be constant at compile-time and hence that GCC can perform constant-folding on expressions involving that value. The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. A return of 0 does not indicate that the value is *not* a constant, but merely that GCC cannot prove it is a constant with the specified value of the '`-O`' option.

You would typically use this function in an embedded application where memory was a critical resource. If you have some complex calculation, you may want it to be folded if it involves constants, but need to call a function if it does not. For example:

```
#define Scale_Value(X)      \
  (__builtin_constant_p (X) \
  ? ((X) * SCALE + OFFSET) : Scale (X))
```

You may use this built-in function in either a macro or an inline function. However, if you use it in an inlined function and pass an argument of the function as the argument to the built-in, GCC will never return 1 when you call the inline function with a string constant or compound literal (see Section 5.21 [Compound Literals], page 168) and will not return 1 when you pass a constant numeric value to the inline function unless you specify the '`-O`' option.

You may also use `__builtin_constant_p` in initializers for static data. For instance, you can write

```
static const int table[] = {
   __builtin_constant_p (EXPRESSION) ? (EXPRESSION) : -1,
   /* ... */
};
```

This is an acceptable initializer even if *EXPRESSION* is not a constant expression. GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

Previous versions of GCC did not accept this built-in in data initializers. The earliest version where it is completely safe is 3.0.1.

`long __builtin_expect (long` *exp*`, long` *c*`)` Built-in Function

You may use `__builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this ('`-fprofile-arcs`'), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of *exp*, which should be an integral expression. The value of *c* must be a compile-time constant. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if (__builtin_expect (x, 0))
  foo ();
```

would indicate that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for *exp*, you should use constructions such as

```
if (__builtin_expect (ptr != NULL, 1))
  error ();
```

when testing pointer or floating-point values.

`void __builtin_prefetch (const void *`*addr*`, ...)` Built-in Function

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` will not fault if `p->next` is not a valid address, but evaluation will fault if `p` is not a valid address.

If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

5.46 Built-in Functions Specific to Particular Target Machines

On some target machines, GCC supports many built-in functions specific to those machines. Generally these generate calls to specific machine instructions, but allow the compiler to schedule those calls.

5.46.1 X86 Built-in Functions

These built-in functions are available for the i386 and x86-64 family of computers, depending on the command-line switches used.

The following machine modes are available for use with MMX built-in functions (see Section 5.44 [Vector Extensions], page 216): `V2SI` for a vector of two 32-bit integers, `V4HI` for a vector of four 16-bit integers, and `V8QI` for a vector of eight 8-bit integers. Some of the built-in functions operate on MMX registers as a whole 64-bit entity, these use `DI` as their mode.

If 3Dnow extensions are enabled, `V2SF` is used as a mode for a vector of two 32-bit floating point values.

If SSE extensions are enabled, `V4SF` is used for a vector of four 32-bit floating point values. Some instructions use a vector of four 32-bit integers, these use `V4SI`. Finally, some instructions operate on an entire vector register, interpreting it as a 128-bit integer, these use mode `TI`.

The following built-in functions are made available by '`-mmmx`'. All of them generate the machine instruction that is part of the name.

```
v8qi __builtin_ia32_paddb (v8qi, v8qi)
v4hi __builtin_ia32_paddw (v4hi, v4hi)
v2si __builtin_ia32_paddd (v2si, v2si)
v8qi __builtin_ia32_psubb (v8qi, v8qi)
v4hi __builtin_ia32_psubw (v4hi, v4hi)
v2si __builtin_ia32_psubd (v2si, v2si)
v8qi __builtin_ia32_paddsb (v8qi, v8qi)
v4hi __builtin_ia32_paddsw (v4hi, v4hi)
v8qi __builtin_ia32_psubsb (v8qi, v8qi)
v4hi __builtin_ia32_psubsw (v4hi, v4hi)
v8qi __builtin_ia32_paddusb (v8qi, v8qi)
v4hi __builtin_ia32_paddusw (v4hi, v4hi)
v8qi __builtin_ia32_psubusb (v8qi, v8qi)
v4hi __builtin_ia32_psubusw (v4hi, v4hi)
v4hi __builtin_ia32_pmullw (v4hi, v4hi)
v4hi __builtin_ia32_pmulhw (v4hi, v4hi)
di __builtin_ia32_pand (di, di)
di __builtin_ia32_pandn (di,di)
di __builtin_ia32_por (di, di)
di __builtin_ia32_pxor (di, di)
v8qi __builtin_ia32_pcmpeqb (v8qi, v8qi)
v4hi __builtin_ia32_pcmpeqw (v4hi, v4hi)
v2si __builtin_ia32_pcmpeqd (v2si, v2si)
v8qi __builtin_ia32_pcmpgtb (v8qi, v8qi)
```

```
v4hi __builtin_ia32_pcmpgtw (v4hi, v4hi)
v2si __builtin_ia32_pcmpgtd (v2si, v2si)
v8qi __builtin_ia32_punpckhbw (v8qi, v8qi)
v4hi __builtin_ia32_punpckhwd (v4hi, v4hi)
v2si __builtin_ia32_punpckhdq (v2si, v2si)
v8qi __builtin_ia32_punpcklbw (v8qi, v8qi)
v4hi __builtin_ia32_punpcklwd (v4hi, v4hi)
v2si __builtin_ia32_punpckldq (v2si, v2si)
v8qi __builtin_ia32_packsswb (v4hi, v4hi)
v4hi __builtin_ia32_packssdw (v2si, v2si)
v8qi __builtin_ia32_packuswb (v4hi, v4hi)
```

The following built-in functions are made available either with '-msse', or with a combination of '-m3dnow' and '-march=athlon'. All of them generate the machine instruction that is part of the name.

```
v4hi __builtin_ia32_pmulhuw (v4hi, v4hi)
v8qi __builtin_ia32_pavgb (v8qi, v8qi)
v4hi __builtin_ia32_pavgw (v4hi, v4hi)
v4hi __builtin_ia32_psadbw (v8qi, v8qi)
v8qi __builtin_ia32_pmaxub (v8qi, v8qi)
v4hi __builtin_ia32_pmaxsw (v4hi, v4hi)
v8qi __builtin_ia32_pminub (v8qi, v8qi)
v4hi __builtin_ia32_pminsw (v4hi, v4hi)
int __builtin_ia32_pextrw (v4hi, int)
v4hi __builtin_ia32_pinsrw (v4hi, int, int)
int __builtin_ia32_pmovmskb (v8qi)
void __builtin_ia32_maskmovq (v8qi, v8qi, char *)
void __builtin_ia32_movntq (di *, di)
void __builtin_ia32_sfence (void)
```

The following built-in functions are available when '-msse' is used. All of them generate the machine instruction that is part of the name.

```
int __builtin_ia32_comieq (v4sf, v4sf)
int __builtin_ia32_comineq (v4sf, v4sf)
int __builtin_ia32_comilt (v4sf, v4sf)
int __builtin_ia32_comile (v4sf, v4sf)
int __builtin_ia32_comigt (v4sf, v4sf)
int __builtin_ia32_comige (v4sf, v4sf)
int __builtin_ia32_ucomieq (v4sf, v4sf)
int __builtin_ia32_ucomineq (v4sf, v4sf)
int __builtin_ia32_ucomilt (v4sf, v4sf)
int __builtin_ia32_ucomile (v4sf, v4sf)
int __builtin_ia32_ucomigt (v4sf, v4sf)
int __builtin_ia32_ucomige (v4sf, v4sf)
v4sf __builtin_ia32_addps (v4sf, v4sf)
v4sf __builtin_ia32_subps (v4sf, v4sf)
v4sf __builtin_ia32_mulps (v4sf, v4sf)
v4sf __builtin_ia32_divps (v4sf, v4sf)
v4sf __builtin_ia32_addss (v4sf, v4sf)
v4sf __builtin_ia32_subss (v4sf, v4sf)
```

```
v4sf __builtin_ia32_mulss (v4sf, v4sf)
v4sf __builtin_ia32_divss (v4sf, v4sf)
v4si __builtin_ia32_cmpeqps (v4sf, v4sf)
v4si __builtin_ia32_cmpltps (v4sf, v4sf)
v4si __builtin_ia32_cmpleps (v4sf, v4sf)
v4si __builtin_ia32_cmpgtps (v4sf, v4sf)
v4si __builtin_ia32_cmpgeps (v4sf, v4sf)
v4si __builtin_ia32_cmpunordps (v4sf, v4sf)
v4si __builtin_ia32_cmpneqps (v4sf, v4sf)
v4si __builtin_ia32_cmpnltps (v4sf, v4sf)
v4si __builtin_ia32_cmpnleps (v4sf, v4sf)
v4si __builtin_ia32_cmpngtps (v4sf, v4sf)
v4si __builtin_ia32_cmpngeps (v4sf, v4sf)
v4si __builtin_ia32_cmpordps (v4sf, v4sf)
v4si __builtin_ia32_cmpeqss (v4sf, v4sf)
v4si __builtin_ia32_cmpltss (v4sf, v4sf)
v4si __builtin_ia32_cmpless (v4sf, v4sf)
v4si __builtin_ia32_cmpgtss (v4sf, v4sf)
v4si __builtin_ia32_cmpgess (v4sf, v4sf)
v4si __builtin_ia32_cmpunordss (v4sf, v4sf)
v4si __builtin_ia32_cmpneqss (v4sf, v4sf)
v4si __builtin_ia32_cmpnlts (v4sf, v4sf)
v4si __builtin_ia32_cmpnless (v4sf, v4sf)
v4si __builtin_ia32_cmpngtss (v4sf, v4sf)
v4si __builtin_ia32_cmpngess (v4sf, v4sf)
v4si __builtin_ia32_cmpordss (v4sf, v4sf)
v4sf __builtin_ia32_maxps (v4sf, v4sf)
v4sf __builtin_ia32_maxss (v4sf, v4sf)
v4sf __builtin_ia32_minps (v4sf, v4sf)
v4sf __builtin_ia32_minss (v4sf, v4sf)
v4sf __builtin_ia32_andps (v4sf, v4sf)
v4sf __builtin_ia32_andnps (v4sf, v4sf)
v4sf __builtin_ia32_orps (v4sf, v4sf)
v4sf __builtin_ia32_xorps (v4sf, v4sf)
v4sf __builtin_ia32_movss (v4sf, v4sf)
v4sf __builtin_ia32_movhlps (v4sf, v4sf)
v4sf __builtin_ia32_movlhps (v4sf, v4sf)
v4sf __builtin_ia32_unpckhps (v4sf, v4sf)
v4sf __builtin_ia32_unpcklps (v4sf, v4sf)
v4sf __builtin_ia32_cvtpi2ps (v4sf, v2si)
v4sf __builtin_ia32_cvtsi2ss (v4sf, int)
v2si __builtin_ia32_cvtps2pi (v4sf)
int __builtin_ia32_cvtss2si (v4sf)
v2si __builtin_ia32_cvttps2pi (v4sf)
int __builtin_ia32_cvttss2si (v4sf)
v4sf __builtin_ia32_rcpps (v4sf)
v4sf __builtin_ia32_rsqrtps (v4sf)
v4sf __builtin_ia32_sqrtps (v4sf)
v4sf __builtin_ia32_rcpss (v4sf)
```

```
v4sf __builtin_ia32_rsqrtss (v4sf)
v4sf __builtin_ia32_sqrtss (v4sf)
v4sf __builtin_ia32_shufps (v4sf, v4sf, int)
void __builtin_ia32_movntps (float *, v4sf)
int __builtin_ia32_movmskps (v4sf)
```

The following built-in functions are available when '-msse' is used.

`v4sf __builtin_ia32_loadaps (float *)`
: Generates the `movaps` machine instruction as a load from memory.

`void __builtin_ia32_storeaps (float *, v4sf)`
: Generates the `movaps` machine instruction as a store to memory.

`v4sf __builtin_ia32_loadups (float *)`
: Generates the `movups` machine instruction as a load from memory.

`void __builtin_ia32_storeups (float *, v4sf)`
: Generates the `movups` machine instruction as a store to memory.

`v4sf __builtin_ia32_loadsss (float *)`
: Generates the `movss` machine instruction as a load from memory.

`void __builtin_ia32_storess (float *, v4sf)`
: Generates the `movss` machine instruction as a store to memory.

`v4sf __builtin_ia32_loadhps (v4sf, v2si *)`
: Generates the `movhps` machine instruction as a load from memory.

`v4sf __builtin_ia32_loadlps (v4sf, v2si *)`
: Generates the `movlps` machine instruction as a load from memory

`void __builtin_ia32_storehps (v4sf, v2si *)`
: Generates the `movhps` machine instruction as a store to memory.

`void __builtin_ia32_storelps (v4sf, v2si *)`
: Generates the `movlps` machine instruction as a store to memory.

The following built-in functions are available when '-m3dnow' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_femms (void)
v8qi __builtin_ia32_pavgusb (v8qi, v8qi)
v2si __builtin_ia32_pf2id (v2sf)
v2sf __builtin_ia32_pfacc (v2sf, v2sf)
v2sf __builtin_ia32_pfadd (v2sf, v2sf)
v2si __builtin_ia32_pfcmpeq (v2sf, v2sf)
v2si __builtin_ia32_pfcmpge (v2sf, v2sf)
v2si __builtin_ia32_pfcmpgt (v2sf, v2sf)
v2sf __builtin_ia32_pfmax (v2sf, v2sf)
v2sf __builtin_ia32_pfmin (v2sf, v2sf)
v2sf __builtin_ia32_pfmul (v2sf, v2sf)
v2sf __builtin_ia32_pfrcp (v2sf)
v2sf __builtin_ia32_pfrcpit1 (v2sf, v2sf)
v2sf __builtin_ia32_pfrcpit2 (v2sf, v2sf)
v2sf __builtin_ia32_pfrsqrt (v2sf)
```

```
v2sf __builtin_ia32_pfrsqrtit1 (v2sf, v2sf)
v2sf __builtin_ia32_pfsub (v2sf, v2sf)
v2sf __builtin_ia32_pfsubr (v2sf, v2sf)
v2sf __builtin_ia32_pi2fd (v2si)
v4hi __builtin_ia32_pmulhrw (v4hi, v4hi)
```

The following built-in functions are available when both '-m3dnow' and '-march=athlon' are used. All of them generate the machine instruction that is part of the name.

```
v2si __builtin_ia32_pf2iw (v2sf)
v2sf __builtin_ia32_pfnacc (v2sf, v2sf)
v2sf __builtin_ia32_pfpnacc (v2sf, v2sf)
v2sf __builtin_ia32_pi2fw (v2si)
v2sf __builtin_ia32_pswapdsf (v2sf)
v2si __builtin_ia32_pswapdsi (v2si)
```

5.46.2 PowerPC AltiVec Built-in Functions

These built-in functions are available for the PowerPC family of computers, depending on the command-line switches used.

The following machine modes are available for use with AltiVec built-in functions (see Section 5.44 [Vector Extensions], page 216): V4SI for a vector of four 32-bit integers, V4SF for a vector of four 32-bit floating point numbers, V8HI for a vector of eight 16-bit integers, and V16QI for a vector of sixteen 8-bit integers.

The following functions are made available by including <altivec.h> and using '-maltivec' and '-mabi=altivec'. The functions implement the functionality described in Motorola's AltiVec Programming Interface Manual.

Note: Only the <altivec.h> interface is supported. Internally, GCC uses built-in functions to achieve the functionality in the aforementioned header file, but they are not supported and are subject to change without notice.

```
vector signed char vec_abs (vector signed char, vector signed char);
vector signed short vec_abs (vector signed short, vector signed short);
vector signed int vec_abs (vector signed int, vector signed int);
vector signed float vec_abs (vector signed float, vector signed float);

vector signed char vec_abss (vector signed char, vector signed char);
vector signed short vec_abss (vector signed short, vector signed short);

vector signed char vec_add (vector signed char, vector signed char);
vector unsigned char vec_add (vector signed char, vector unsigned char);

vector unsigned char vec_add (vector unsigned char, vector signed char);

vector unsigned char vec_add (vector unsigned char,
                              vector unsigned char);
vector signed short vec_add (vector signed short, vector signed short);
vector unsigned short vec_add (vector signed short,
                               vector unsigned short);
vector unsigned short vec_add (vector unsigned short,
```

```
                               vector signed short);
vector unsigned short vec_add (vector unsigned short,
                               vector unsigned short);
vector signed int vec_add (vector signed int, vector signed int);
vector unsigned int vec_add (vector signed int, vector unsigned int);
vector unsigned int vec_add (vector unsigned int, vector signed int);
vector unsigned int vec_add (vector unsigned int, vector unsigned int);
vector float vec_add (vector float, vector float);

vector unsigned int vec_addc (vector unsigned int, vector unsigned int);

vector unsigned char vec_adds (vector signed char,
                               vector unsigned char);
vector unsigned char vec_adds (vector unsigned char,
                               vector signed char);
vector unsigned char vec_adds (vector unsigned char,
                               vector unsigned char);
vector signed char vec_adds (vector signed char, vector signed char);
vector unsigned short vec_adds (vector signed short,
                                vector unsigned short);
vector unsigned short vec_adds (vector unsigned short,
                                vector signed short);
vector unsigned short vec_adds (vector unsigned short,
                                vector unsigned short);
vector signed short vec_adds (vector signed short, vector signed short);

vector unsigned int vec_adds (vector signed int, vector unsigned int);
vector unsigned int vec_adds (vector unsigned int, vector signed int);
vector unsigned int vec_adds (vector unsigned int, vector unsigned int);

vector signed int vec_adds (vector signed int, vector signed int);

vector float vec_and (vector float, vector float);
vector float vec_and (vector float, vector signed int);
vector float vec_and (vector signed int, vector float);
vector signed int vec_and (vector signed int, vector signed int);
vector unsigned int vec_and (vector signed int, vector unsigned int);
vector unsigned int vec_and (vector unsigned int, vector signed int);
vector unsigned int vec_and (vector unsigned int, vector unsigned int);
vector signed short vec_and (vector signed short, vector signed short);
vector unsigned short vec_and (vector signed short,
                               vector unsigned short);
vector unsigned short vec_and (vector unsigned short,
                               vector signed short);
vector unsigned short vec_and (vector unsigned short,
                               vector unsigned short);
vector signed char vec_and (vector signed char, vector signed char);
vector unsigned char vec_and (vector signed char, vector unsigned char);
```

```
vector unsigned char vec_and (vector unsigned char, vector signed char);

vector unsigned char vec_and (vector unsigned char,
                              vector unsigned char);

vector float vec_andc (vector float, vector float);
vector float vec_andc (vector float, vector signed int);
vector float vec_andc (vector signed int, vector float);
vector signed int vec_andc (vector signed int, vector signed int);
vector unsigned int vec_andc (vector signed int, vector unsigned int);
vector unsigned int vec_andc (vector unsigned int, vector signed int);
vector unsigned int vec_andc (vector unsigned int, vector unsigned int);

vector signed short vec_andc (vector signed short, vector signed short);

vector unsigned short vec_andc (vector signed short,
                                vector unsigned short);
vector unsigned short vec_andc (vector unsigned short,
                                vector signed short);
vector unsigned short vec_andc (vector unsigned short,
                                vector unsigned short);
vector signed char vec_andc (vector signed char, vector signed char);
vector unsigned char vec_andc (vector signed char,
                               vector unsigned char);
vector unsigned char vec_andc (vector unsigned char,
                               vector signed char);
vector unsigned char vec_andc (vector unsigned char,
                               vector unsigned char);

vector unsigned char vec_avg (vector unsigned char,
                              vector unsigned char);
vector signed char vec_avg (vector signed char, vector signed char);
vector unsigned short vec_avg (vector unsigned short,
                               vector unsigned short);
vector signed short vec_avg (vector signed short, vector signed short);
vector unsigned int vec_avg (vector unsigned int, vector unsigned int);
vector signed int vec_avg (vector signed int, vector signed int);

vector float vec_ceil (vector float);

vector signed int vec_cmpb (vector float, vector float);

vector signed char vec_cmpeq (vector signed char, vector signed char);
vector signed char vec_cmpeq (vector unsigned char,
                              vector unsigned char);
vector signed short vec_cmpeq (vector signed short,
                               vector signed short);
vector signed short vec_cmpeq (vector unsigned short,
                               vector unsigned short);
```

```
vector signed int vec_cmpeq (vector signed int, vector signed int);
vector signed int vec_cmpeq (vector unsigned int, vector unsigned int);
vector signed int vec_cmpeq (vector float, vector float);

vector signed int vec_cmpge (vector float, vector float);

vector signed char vec_cmpgt (vector unsigned char,
                              vector unsigned char);
vector signed char vec_cmpgt (vector signed char, vector signed char);
vector signed short vec_cmpgt (vector unsigned short,
                               vector unsigned short);
vector signed short vec_cmpgt (vector signed short,
                               vector signed short);
vector signed int vec_cmpgt (vector unsigned int, vector unsigned int);
vector signed int vec_cmpgt (vector signed int, vector signed int);
vector signed int vec_cmpgt (vector float, vector float);

vector signed int vec_cmple (vector float, vector float);

vector signed char vec_cmplt (vector unsigned char,
                              vector unsigned char);
vector signed char vec_cmplt (vector signed char, vector signed char);
vector signed short vec_cmplt (vector unsigned short,
                               vector unsigned short);
vector signed short vec_cmplt (vector signed short,
                               vector signed short);
vector signed int vec_cmplt (vector unsigned int, vector unsigned int);
vector signed int vec_cmplt (vector signed int, vector signed int);
vector signed int vec_cmplt (vector float, vector float);

vector float vec_ctf (vector unsigned int, const char);
vector float vec_ctf (vector signed int, const char);

vector signed int vec_cts (vector float, const char);

vector unsigned int vec_ctu (vector float, const char);

void vec_dss (const char);

void vec_dssall (void);

void vec_dst (void *, int, const char);

void vec_dstst (void *, int, const char);

void vec_dststt (void *, int, const char);

void vec_dstt (void *, int, const char);
```

```
vector float vec_expte (vector float, vector float);

vector float vec_floor (vector float, vector float);

vector float vec_ld (int, vector float *);
vector float vec_ld (int, float *):
vector signed int vec_ld (int, int *);
vector signed int vec_ld (int, vector signed int *);
vector unsigned int vec_ld (int, vector unsigned int *);
vector unsigned int vec_ld (int, unsigned int *);
vector signed short vec_ld (int, short *, vector signed short *);
vector unsigned short vec_ld (int, unsigned short *,
                              vector unsigned short *);
vector signed char vec_ld (int, signed char *);
vector signed char vec_ld (int, vector signed char *);
vector unsigned char vec_ld (int, unsigned char *);
vector unsigned char vec_ld (int, vector unsigned char *);

vector signed char vec_lde (int, signed char *);
vector unsigned char vec_lde (int, unsigned char *);
vector signed short vec_lde (int, short *);
vector unsigned short vec_lde (int, unsigned short *);
vector float vec_lde (int, float *);
vector signed int vec_lde (int, int *);
vector unsigned int vec_lde (int, unsigned int *);

void float vec_ldl (int, float *);
void float vec_ldl (int, vector float *);
void signed int vec_ldl (int, vector signed int *);
void signed int vec_ldl (int, int *);
void unsigned int vec_ldl (int, unsigned int *);
void unsigned int vec_ldl (int, vector unsigned int *);
void signed short vec_ldl (int, vector signed short *);
void signed short vec_ldl (int, short *);
void unsigned short vec_ldl (int, vector unsigned short *);
void unsigned short vec_ldl (int, unsigned short *);
void signed char vec_ldl (int, vector signed char *);
void signed char vec_ldl (int, signed char *);
void unsigned char vec_ldl (int, vector unsigned char *);
void unsigned char vec_ldl (int, unsigned char *);

vector float vec_loge (vector float);

vector unsigned char vec_lvsl (int, void *, int *);

vector unsigned char vec_lvsr (int, void *, int *);

vector float vec_madd (vector float, vector float, vector float);
```

```
vector signed short vec_madds (vector signed short, vector signed short,
                               vector signed short);

vector unsigned char vec_max (vector signed char, vector unsigned char);

vector unsigned char vec_max (vector unsigned char, vector signed char);

vector unsigned char vec_max (vector unsigned char,
                              vector unsigned char);
vector signed char vec_max (vector signed char, vector signed char);
vector unsigned short vec_max (vector signed short,
                               vector unsigned short);
vector unsigned short vec_max (vector unsigned short,
                               vector signed short);
vector unsigned short vec_max (vector unsigned short,
                               vector unsigned short);
vector signed short vec_max (vector signed short, vector signed short);
vector unsigned int vec_max (vector signed int, vector unsigned int);
vector unsigned int vec_max (vector unsigned int, vector signed int);
vector unsigned int vec_max (vector unsigned int, vector unsigned int);
vector signed int vec_max (vector signed int, vector signed int);
vector float vec_max (vector float, vector float);

vector signed char vec_mergeh (vector signed char, vector signed char);
vector unsigned char vec_mergeh (vector unsigned char,
                                 vector unsigned char);
vector signed short vec_mergeh (vector signed short,
                                vector signed short);
vector unsigned short vec_mergeh (vector unsigned short,
                                  vector unsigned short);
vector float vec_mergeh (vector float, vector float);
vector signed int vec_mergeh (vector signed int, vector signed int);
vector unsigned int vec_mergeh (vector unsigned int,
                                vector unsigned int);

vector signed char vec_mergel (vector signed char, vector signed char);
vector unsigned char vec_mergel (vector unsigned char,
                                 vector unsigned char);
vector signed short vec_mergel (vector signed short,
                                vector signed short);
vector unsigned short vec_mergel (vector unsigned short,
                                  vector unsigned short);
vector float vec_mergel (vector float, vector float);
vector signed int vec_mergel (vector signed int, vector signed int);
vector unsigned int vec_mergel (vector unsigned int,
                                vector unsigned int);

vector unsigned short vec_mfvscr (void);
```

```
vector unsigned char vec_min (vector signed char, vector unsigned char);

vector unsigned char vec_min (vector unsigned char, vector signed char);

vector unsigned char vec_min (vector unsigned char,
                              vector unsigned char);
vector signed char vec_min (vector signed char, vector signed char);
vector unsigned short vec_min (vector signed short,
                               vector unsigned short);
vector unsigned short vec_min (vector unsigned short,
                               vector signed short);
vector unsigned short vec_min (vector unsigned short,
                               vector unsigned short);
vector signed short vec_min (vector signed short, vector signed short);
vector unsigned int vec_min (vector signed int, vector unsigned int);
vector unsigned int vec_min (vector unsigned int, vector signed int);
vector unsigned int vec_min (vector unsigned int, vector unsigned int);
vector signed int vec_min (vector signed int, vector signed int);
vector float vec_min (vector float, vector float);

vector signed short vec_mladd (vector signed short, vector signed short,
                               vector signed short);
vector signed short vec_mladd (vector signed short,
                               vector unsigned short,
                               vector unsigned short);
vector signed short vec_mladd (vector unsigned short,
                               vector signed short,
                               vector signed short);
vector unsigned short vec_mladd (vector unsigned short,
                                 vector unsigned short,
                                 vector unsigned short);

vector signed short vec_mradds (vector signed short,
                                vector signed short,
                                vector signed short);

vector unsigned int vec_msum (vector unsigned char,
                              vector unsigned char,
                              vector unsigned int);
vector signed int vec_msum (vector signed char, vector unsigned char,
                            vector signed int);
vector unsigned int vec_msum (vector unsigned short,
                              vector unsigned short,
                              vector unsigned int);
vector signed int vec_msum (vector signed short, vector signed short,
                            vector signed int);

vector unsigned int vec_msums (vector unsigned short,
                               vector unsigned short,
```

```
                              vector unsigned int);
vector signed int vec_msums (vector signed short, vector signed short,
                             vector signed int);

void vec_mtvscr (vector signed int);
void vec_mtvscr (vector unsigned int);
void vec_mtvscr (vector signed short);
void vec_mtvscr (vector unsigned short);
void vec_mtvscr (vector signed char);
void vec_mtvscr (vector unsigned char);

vector unsigned short vec_mule (vector unsigned char,
                                vector unsigned char);
vector signed short vec_mule (vector signed char, vector signed char);
vector unsigned int vec_mule (vector unsigned short,
                              vector unsigned short);
vector signed int vec_mule (vector signed short, vector signed short);

vector unsigned short vec_mulo (vector unsigned char,
                                vector unsigned char);
vector signed short vec_mulo (vector signed char, vector signed char);
vector unsigned int vec_mulo (vector unsigned short,
                              vector unsigned short);
vector signed int vec_mulo (vector signed short, vector signed short);

vector float vec_nmsub (vector float, vector float, vector float);

vector float vec_nor (vector float, vector float);
vector signed int vec_nor (vector signed int, vector signed int);
vector unsigned int vec_nor (vector unsigned int, vector unsigned int);
vector signed short vec_nor (vector signed short, vector signed short);
vector unsigned short vec_nor (vector unsigned short,
                               vector unsigned short);
vector signed char vec_nor (vector signed char, vector signed char);
vector unsigned char vec_nor (vector unsigned char,
                              vector unsigned char);

vector float vec_or (vector float, vector float);
vector float vec_or (vector float, vector signed int);
vector float vec_or (vector signed int, vector float);
vector signed int vec_or (vector signed int, vector signed int);
vector unsigned int vec_or (vector signed int, vector unsigned int);
vector unsigned int vec_or (vector unsigned int, vector signed int);
vector unsigned int vec_or (vector unsigned int, vector unsigned int);
vector signed short vec_or (vector signed short, vector signed short);
vector unsigned short vec_or (vector signed short,
                              vector unsigned short);
vector unsigned short vec_or (vector unsigned short,
                              vector signed short);
```

```
vector unsigned short vec_or (vector unsigned short,
                              vector unsigned short);
vector signed char vec_or (vector signed char, vector signed char);
vector unsigned char vec_or (vector signed char, vector unsigned char);
vector unsigned char vec_or (vector unsigned char, vector signed char);
vector unsigned char vec_or (vector unsigned char,
                             vector unsigned char);

vector signed char vec_pack (vector signed short, vector signed short);
vector unsigned char vec_pack (vector unsigned short,
                               vector unsigned short);
vector signed short vec_pack (vector signed int, vector signed int);
vector unsigned short vec_pack (vector unsigned int,
                                vector unsigned int);

vector signed short vec_packpx (vector unsigned int,
                                vector unsigned int);

vector unsigned char vec_packs (vector unsigned short,
                                vector unsigned short);
vector signed char vec_packs (vector signed short, vector signed short);

vector unsigned short vec_packs (vector unsigned int,
                                 vector unsigned int);
vector signed short vec_packs (vector signed int, vector signed int);

vector unsigned char vec_packsu (vector unsigned short,
                                 vector unsigned short);
vector unsigned char vec_packsu (vector signed short,
                                 vector signed short);
vector unsigned short vec_packsu (vector unsigned int,
                                  vector unsigned int);
vector unsigned short vec_packsu (vector signed int, vector signed int);

vector float vec_perm (vector float, vector float,
                       vector unsigned char);
vector signed int vec_perm (vector signed int, vector signed int,
                            vector unsigned char);
vector unsigned int vec_perm (vector unsigned int, vector unsigned int,
                              vector unsigned char);
vector signed short vec_perm (vector signed short, vector signed short,
                              vector unsigned char);
vector unsigned short vec_perm (vector unsigned short,
                                vector unsigned short,
                                vector unsigned char);
vector signed char vec_perm (vector signed char, vector signed char,
                             vector unsigned char);
vector unsigned char vec_perm (vector unsigned char,
                               vector unsigned char,
```

```
                              vector unsigned char);

vector float vec_re (vector float);

vector signed char vec_rl (vector signed char, vector unsigned char);
vector unsigned char vec_rl (vector unsigned char,
                             vector unsigned char);
vector signed short vec_rl (vector signed short, vector unsigned short);

vector unsigned short vec_rl (vector unsigned short,
                              vector unsigned short);
vector signed int vec_rl (vector signed int, vector unsigned int);
vector unsigned int vec_rl (vector unsigned int, vector unsigned int);

vector float vec_round (vector float);

vector float vec_rsqrte (vector float);

vector float vec_sel (vector float, vector float, vector signed int);
vector float vec_sel (vector float, vector float, vector unsigned int);
vector signed int vec_sel (vector signed int, vector signed int,
                           vector signed int);
vector signed int vec_sel (vector signed int, vector signed int,
                           vector unsigned int);
vector unsigned int vec_sel (vector unsigned int, vector unsigned int,
                             vector signed int);
vector unsigned int vec_sel (vector unsigned int, vector unsigned int,
                             vector unsigned int);
vector signed short vec_sel (vector signed short, vector signed short,
                             vector signed short);
vector signed short vec_sel (vector signed short, vector signed short,
                             vector unsigned short);
vector unsigned short vec_sel (vector unsigned short,
                               vector unsigned short,
                               vector signed short);
vector unsigned short vec_sel (vector unsigned short,
                               vector unsigned short,
                               vector unsigned short);
vector signed char vec_sel (vector signed char, vector signed char,
                            vector signed char);
vector signed char vec_sel (vector signed char, vector signed char,
                            vector unsigned char);
vector unsigned char vec_sel (vector unsigned char,
                              vector unsigned char,
                              vector signed char);
vector unsigned char vec_sel (vector unsigned char,
                              vector unsigned char,
                              vector unsigned char);
```

```
vector signed char vec_sl (vector signed char, vector unsigned char);
vector unsigned char vec_sl (vector unsigned char,
                             vector unsigned char);
vector signed short vec_sl (vector signed short, vector unsigned short);

vector unsigned short vec_sl (vector unsigned short,
                              vector unsigned short);
vector signed int vec_sl (vector signed int, vector unsigned int);
vector unsigned int vec_sl (vector unsigned int, vector unsigned int);

vector float vec_sld (vector float, vector float, const char);
vector signed int vec_sld (vector signed int, vector signed int,
                           const char);
vector unsigned int vec_sld (vector unsigned int, vector unsigned int,
                             const char);
vector signed short vec_sld (vector signed short, vector signed short,
                             const char);
vector unsigned short vec_sld (vector unsigned short,
                               vector unsigned short, const char);
vector signed char vec_sld (vector signed char, vector signed char,
                            const char);
vector unsigned char vec_sld (vector unsigned char,
                              vector unsigned char,
                              const char);

vector signed int vec_sll (vector signed int, vector unsigned int);
vector signed int vec_sll (vector signed int, vector unsigned short);
vector signed int vec_sll (vector signed int, vector unsigned char);
vector unsigned int vec_sll (vector unsigned int, vector unsigned int);
vector unsigned int vec_sll (vector unsigned int,
                             vector unsigned short);
vector unsigned int vec_sll (vector unsigned int, vector unsigned char);

vector signed short vec_sll (vector signed short, vector unsigned int);
vector signed short vec_sll (vector signed short,
                             vector unsigned short);
vector signed short vec_sll (vector signed short, vector unsigned char);

vector unsigned short vec_sll (vector unsigned short,
                               vector unsigned int);
vector unsigned short vec_sll (vector unsigned short,
                               vector unsigned short);
vector unsigned short vec_sll (vector unsigned short,
                               vector unsigned char);
vector signed char vec_sll (vector signed char, vector unsigned int);
vector signed char vec_sll (vector signed char, vector unsigned short);
vector signed char vec_sll (vector signed char, vector unsigned char);
vector unsigned char vec_sll (vector unsigned char,
                              vector unsigned int);
```

```
vector unsigned char vec_sll (vector unsigned char,
                              vector unsigned short);
vector unsigned char vec_sll (vector unsigned char,
                              vector unsigned char);

vector float vec_slo (vector float, vector signed char);
vector float vec_slo (vector float, vector unsigned char);
vector signed int vec_slo (vector signed int, vector signed char);
vector signed int vec_slo (vector signed int, vector unsigned char);
vector unsigned int vec_slo (vector unsigned int, vector signed char);
vector unsigned int vec_slo (vector unsigned int, vector unsigned char);

vector signed short vec_slo (vector signed short, vector signed char);
vector signed short vec_slo (vector signed short, vector unsigned char);

vector unsigned short vec_slo (vector unsigned short,
                               vector signed char);
vector unsigned short vec_slo (vector unsigned short,
                               vector unsigned char);
vector signed char vec_slo (vector signed char, vector signed char);
vector signed char vec_slo (vector signed char, vector unsigned char);
vector unsigned char vec_slo (vector unsigned char, vector signed char);

vector unsigned char vec_slo (vector unsigned char,
                              vector unsigned char);

vector signed char vec_splat (vector signed char, const char);
vector unsigned char vec_splat (vector unsigned char, const char);
vector signed short vec_splat (vector signed short, const char);
vector unsigned short vec_splat (vector unsigned short, const char);
vector float vec_splat (vector float, const char);
vector signed int vec_splat (vector signed int, const char);
vector unsigned int vec_splat (vector unsigned int, const char);

vector signed char vec_splat_s8 (const char);

vector signed short vec_splat_s16 (const char);

vector signed int vec_splat_s32 (const char);

vector unsigned char vec_splat_u8 (const char);

vector unsigned short vec_splat_u16 (const char);

vector unsigned int vec_splat_u32 (const char);

vector signed char vec_sr (vector signed char, vector unsigned char);
vector unsigned char vec_sr (vector unsigned char,
                             vector unsigned char);
```

```
vector signed short vec_sr (vector signed short, vector unsigned short);

vector unsigned short vec_sr (vector unsigned short,
                              vector unsigned short);
vector signed int vec_sr (vector signed int, vector unsigned int);
vector unsigned int vec_sr (vector unsigned int, vector unsigned int);

vector signed char vec_sra (vector signed char, vector unsigned char);
vector unsigned char vec_sra (vector unsigned char,
                              vector unsigned char);
vector signed short vec_sra (vector signed short,
                             vector unsigned short);
vector unsigned short vec_sra (vector unsigned short,
                               vector unsigned short);
vector signed int vec_sra (vector signed int, vector unsigned int);
vector unsigned int vec_sra (vector unsigned int, vector unsigned int);

vector signed int vec_srl (vector signed int, vector unsigned int);
vector signed int vec_srl (vector signed int, vector unsigned short);
vector signed int vec_srl (vector signed int, vector unsigned char);
vector unsigned int vec_srl (vector unsigned int, vector unsigned int);
vector unsigned int vec_srl (vector unsigned int,
                             vector unsigned short);
vector unsigned int vec_srl (vector unsigned int, vector unsigned char);

vector signed short vec_srl (vector signed short, vector unsigned int);
vector signed short vec_srl (vector signed short,
                             vector unsigned short);
vector signed short vec_srl (vector signed short, vector unsigned char);

vector unsigned short vec_srl (vector unsigned short,
                               vector unsigned int);
vector unsigned short vec_srl (vector unsigned short,
                               vector unsigned short);
vector unsigned short vec_srl (vector unsigned short,
                               vector unsigned char);
vector signed char vec_srl (vector signed char, vector unsigned int);
vector signed char vec_srl (vector signed char, vector unsigned short);
vector signed char vec_srl (vector signed char, vector unsigned char);
vector unsigned char vec_srl (vector unsigned char,
                              vector unsigned int);
vector unsigned char vec_srl (vector unsigned char,
                              vector unsigned short);
vector unsigned char vec_srl (vector unsigned char,
                              vector unsigned char);

vector float vec_sro (vector float, vector signed char);
vector float vec_sro (vector float, vector unsigned char);
vector signed int vec_sro (vector signed int, vector signed char);
```

```
vector signed int vec_sro (vector signed int, vector unsigned char);
vector unsigned int vec_sro (vector unsigned int, vector signed char);
vector unsigned int vec_sro (vector unsigned int, vector unsigned char);

vector signed short vec_sro (vector signed short, vector signed char);
vector signed short vec_sro (vector signed short, vector unsigned char);

vector unsigned short vec_sro (vector unsigned short,
                               vector signed char);
vector unsigned short vec_sro (vector unsigned short,
                               vector unsigned char);
vector signed char vec_sro (vector signed char, vector signed char);
vector signed char vec_sro (vector signed char, vector unsigned char);
vector unsigned char vec_sro (vector unsigned char, vector signed char);

vector unsigned char vec_sro (vector unsigned char,
                              vector unsigned char);

void vec_st (vector float, int, float *);
void vec_st (vector float, int, vector float *);
void vec_st (vector signed int, int, int *);
void vec_st (vector signed int, int, unsigned int *);
void vec_st (vector unsigned int, int, unsigned int *);
void vec_st (vector unsigned int, int, vector unsigned int *);
void vec_st (vector signed short, int, short *);
void vec_st (vector signed short, int, vector unsigned short *);
void vec_st (vector signed short, int, vector signed short *);
void vec_st (vector unsigned short, int, unsigned short *);
void vec_st (vector unsigned short, int, vector unsigned short *);
void vec_st (vector signed char, int, signed char *);
void vec_st (vector signed char, int, unsigned char *);
void vec_st (vector signed char, int, vector signed char *);
void vec_st (vector unsigned char, int, unsigned char *);
void vec_st (vector unsigned char, int, vector unsigned char *);

void vec_ste (vector signed char, int, unsigned char *);
void vec_ste (vector signed char, int, signed char *);
void vec_ste (vector unsigned char, int, unsigned char *);
void vec_ste (vector signed short, int, short *);
void vec_ste (vector signed short, int, unsigned short *);
void vec_ste (vector unsigned short, int, void *);
void vec_ste (vector signed int, int, unsigned int *);
void vec_ste (vector signed int, int, int *);
void vec_ste (vector unsigned int, int, unsigned int *);
void vec_ste (vector float, int, float *);

void vec_stl (vector float, int, vector float *);
void vec_stl (vector float, int, float *);
void vec_stl (vector signed int, int, vector signed int *);
```

```
void vec_stl (vector signed int, int, int *);
void vec_stl (vector signed int, int, unsigned int *);
void vec_stl (vector unsigned int, int, vector unsigned int *);
void vec_stl (vector unsigned int, int, unsigned int *);
void vec_stl (vector signed short, int, short *);
void vec_stl (vector signed short, int, unsigned short *);
void vec_stl (vector signed short, int, vector signed short *);
void vec_stl (vector unsigned short, int, unsigned short *);
void vec_stl (vector unsigned short, int, vector signed short *);
void vec_stl (vector signed char, int, signed char *);
void vec_stl (vector signed char, int, unsigned char *);
void vec_stl (vector signed char, int, vector signed char *);
void vec_stl (vector unsigned char, int, unsigned char *);
void vec_stl (vector unsigned char, int, vector unsigned char *);

vector signed char vec_sub (vector signed char, vector signed char);
vector unsigned char vec_sub (vector signed char, vector unsigned char);

vector unsigned char vec_sub (vector unsigned char, vector signed char);

vector unsigned char vec_sub (vector unsigned char,
                              vector unsigned char);
vector signed short vec_sub (vector signed short, vector signed short);
vector unsigned short vec_sub (vector signed short,
                               vector unsigned short);
vector unsigned short vec_sub (vector unsigned short,
                               vector signed short);
vector unsigned short vec_sub (vector unsigned short,
                               vector unsigned short);
vector signed int vec_sub (vector signed int, vector signed int);
vector unsigned int vec_sub (vector signed int, vector unsigned int);
vector unsigned int vec_sub (vector unsigned int, vector signed int);
vector unsigned int vec_sub (vector unsigned int, vector unsigned int);
vector float vec_sub (vector float, vector float);

vector unsigned int vec_subc (vector unsigned int, vector unsigned int);

vector unsigned char vec_subs (vector signed char,
                               vector unsigned char);
vector unsigned char vec_subs (vector unsigned char,
                               vector signed char);
vector unsigned char vec_subs (vector unsigned char,
                               vector unsigned char);
vector signed char vec_subs (vector signed char, vector signed char);
vector unsigned short vec_subs (vector signed short,
                                vector unsigned short);
vector unsigned short vec_subs (vector unsigned short,
                                vector signed short);
vector unsigned short vec_subs (vector unsigned short,
```

```
                                vector unsigned short);
vector signed short vec_subs (vector signed short, vector signed short);

vector unsigned int vec_subs (vector signed int, vector unsigned int);
vector unsigned int vec_subs (vector unsigned int, vector signed int);
vector unsigned int vec_subs (vector unsigned int, vector unsigned int);

vector signed int vec_subs (vector signed int, vector signed int);

vector unsigned int vec_sum4s (vector unsigned char,
                               vector unsigned int);
vector signed int vec_sum4s (vector signed char, vector signed int);
vector signed int vec_sum4s (vector signed short, vector signed int);

vector signed int vec_sum2s (vector signed int, vector signed int);

vector signed int vec_sums (vector signed int, vector signed int);

vector float vec_trunc (vector float);

vector signed short vec_unpackh (vector signed char);
vector unsigned int vec_unpackh (vector signed short);
vector signed int vec_unpackh (vector signed short);

vector signed short vec_unpackl (vector signed char);
vector unsigned int vec_unpackl (vector signed short);
vector signed int vec_unpackl (vector signed short);

vector float vec_xor (vector float, vector float);
vector float vec_xor (vector float, vector signed int);
vector float vec_xor (vector signed int, vector float);
vector signed int vec_xor (vector signed int, vector signed int);
vector unsigned int vec_xor (vector signed int, vector unsigned int);
vector unsigned int vec_xor (vector unsigned int, vector signed int);
vector unsigned int vec_xor (vector unsigned int, vector unsigned int);
vector signed short vec_xor (vector signed short, vector signed short);
vector unsigned short vec_xor (vector signed short,
                               vector unsigned short);
vector unsigned short vec_xor (vector unsigned short,
                               vector signed short);
vector unsigned short vec_xor (vector unsigned short,
                               vector unsigned short);
vector signed char vec_xor (vector signed char, vector signed char);
vector unsigned char vec_xor (vector signed char, vector unsigned char);

vector unsigned char vec_xor (vector unsigned char, vector signed char);

vector unsigned char vec_xor (vector unsigned char,
                              vector unsigned char);
```

```
vector signed int vec_all_eq (vector signed char, vector unsigned char);

vector signed int vec_all_eq (vector signed char, vector signed char);
vector signed int vec_all_eq (vector unsigned char, vector signed char);

vector signed int vec_all_eq (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_eq (vector signed short,
                              vector unsigned short);
vector signed int vec_all_eq (vector signed short, vector signed short);

vector signed int vec_all_eq (vector unsigned short,
                              vector signed short);
vector signed int vec_all_eq (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_eq (vector signed int, vector unsigned int);
vector signed int vec_all_eq (vector signed int, vector signed int);
vector signed int vec_all_eq (vector unsigned int, vector signed int);
vector signed int vec_all_eq (vector unsigned int, vector unsigned int);

vector signed int vec_all_eq (vector float, vector float);

vector signed int vec_all_ge (vector signed char, vector unsigned char);

vector signed int vec_all_ge (vector unsigned char, vector signed char);

vector signed int vec_all_ge (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_ge (vector signed char, vector signed char);
vector signed int vec_all_ge (vector signed short,
                              vector unsigned short);
vector signed int vec_all_ge (vector unsigned short,
                              vector signed short);
vector signed int vec_all_ge (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_ge (vector signed short, vector signed short);

vector signed int vec_all_ge (vector signed int, vector unsigned int);
vector signed int vec_all_ge (vector unsigned int, vector signed int);
vector signed int vec_all_ge (vector unsigned int, vector unsigned int);

vector signed int vec_all_ge (vector signed int, vector signed int);
vector signed int vec_all_ge (vector float, vector float);

vector signed int vec_all_gt (vector signed char, vector unsigned char);

vector signed int vec_all_gt (vector unsigned char, vector signed char);
```

```
vector signed int vec_all_gt (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_gt (vector signed char, vector signed char);
vector signed int vec_all_gt (vector signed short,
                              vector unsigned short);
vector signed int vec_all_gt (vector unsigned short,
                              vector signed short);
vector signed int vec_all_gt (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_gt (vector signed short, vector signed short);

vector signed int vec_all_gt (vector signed int, vector unsigned int);
vector signed int vec_all_gt (vector unsigned int, vector signed int);
vector signed int vec_all_gt (vector unsigned int, vector unsigned int);

vector signed int vec_all_gt (vector signed int, vector signed int);
vector signed int vec_all_gt (vector float, vector float);

vector signed int vec_all_in (vector float, vector float);

vector signed int vec_all_le (vector signed char, vector unsigned char);

vector signed int vec_all_le (vector unsigned char, vector signed char);

vector signed int vec_all_le (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_le (vector signed char, vector signed char);
vector signed int vec_all_le (vector signed short,
                              vector unsigned short);
vector signed int vec_all_le (vector unsigned short,
                              vector signed short);
vector signed int vec_all_le (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_le (vector signed short, vector signed short);

vector signed int vec_all_le (vector signed int, vector unsigned int);
vector signed int vec_all_le (vector unsigned int, vector signed int);
vector signed int vec_all_le (vector unsigned int, vector unsigned int);

vector signed int vec_all_le (vector signed int, vector signed int);
vector signed int vec_all_le (vector float, vector float);

vector signed int vec_all_lt (vector signed char, vector unsigned char);

vector signed int vec_all_lt (vector unsigned char, vector signed char);

vector signed int vec_all_lt (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_lt (vector signed char, vector signed char);
```

```
vector signed int vec_all_lt (vector signed short,
                              vector unsigned short);
vector signed int vec_all_lt (vector unsigned short,
                              vector signed short);
vector signed int vec_all_lt (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_lt (vector signed short, vector signed short);

vector signed int vec_all_lt (vector signed int, vector unsigned int);
vector signed int vec_all_lt (vector unsigned int, vector signed int);
vector signed int vec_all_lt (vector unsigned int, vector unsigned int);

vector signed int vec_all_lt (vector signed int, vector signed int);
vector signed int vec_all_lt (vector float, vector float);

vector signed int vec_all_nan (vector float);

vector signed int vec_all_ne (vector signed char, vector unsigned char);

vector signed int vec_all_ne (vector signed char, vector signed char);
vector signed int vec_all_ne (vector unsigned char, vector signed char);

vector signed int vec_all_ne (vector unsigned char,
                              vector unsigned char);
vector signed int vec_all_ne (vector signed short,
                              vector unsigned short);
vector signed int vec_all_ne (vector signed short, vector signed short);

vector signed int vec_all_ne (vector unsigned short,
                              vector signed short);
vector signed int vec_all_ne (vector unsigned short,
                              vector unsigned short);
vector signed int vec_all_ne (vector signed int, vector unsigned int);
vector signed int vec_all_ne (vector signed int, vector signed int);
vector signed int vec_all_ne (vector unsigned int, vector signed int);
vector signed int vec_all_ne (vector unsigned int, vector unsigned int);

vector signed int vec_all_ne (vector float, vector float);

vector signed int vec_all_nge (vector float, vector float);

vector signed int vec_all_ngt (vector float, vector float);

vector signed int vec_all_nle (vector float, vector float);

vector signed int vec_all_nlt (vector float, vector float);

vector signed int vec_all_numeric (vector float);
```

```
vector signed int vec_any_eq (vector signed char, vector unsigned char);

vector signed int vec_any_eq (vector signed char, vector signed char);
vector signed int vec_any_eq (vector unsigned char, vector signed char);

vector signed int vec_any_eq (vector unsigned char,
                              vector unsigned char);
vector signed int vec_any_eq (vector signed short,
                              vector unsigned short);
vector signed int vec_any_eq (vector signed short, vector signed short);

vector signed int vec_any_eq (vector unsigned short,
                              vector signed short);
vector signed int vec_any_eq (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_eq (vector signed int, vector unsigned int);
vector signed int vec_any_eq (vector signed int, vector signed int);
vector signed int vec_any_eq (vector unsigned int, vector signed int);
vector signed int vec_any_eq (vector unsigned int, vector unsigned int);

vector signed int vec_any_eq (vector float, vector float);

vector signed int vec_any_ge (vector signed char, vector unsigned char);

vector signed int vec_any_ge (vector unsigned char, vector signed char);

vector signed int vec_any_ge (vector unsigned char,
                              vector unsigned char);
vector signed int vec_any_ge (vector signed char, vector signed char);
vector signed int vec_any_ge (vector signed short,
                              vector unsigned short);
vector signed int vec_any_ge (vector unsigned short,
                              vector signed short);
vector signed int vec_any_ge (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_ge (vector signed short, vector signed short);

vector signed int vec_any_ge (vector signed int, vector unsigned int);
vector signed int vec_any_ge (vector unsigned int, vector signed int);
vector signed int vec_any_ge (vector unsigned int, vector unsigned int);

vector signed int vec_any_ge (vector signed int, vector signed int);
vector signed int vec_any_ge (vector float, vector float);

vector signed int vec_any_gt (vector signed char, vector unsigned char);

vector signed int vec_any_gt (vector unsigned char, vector signed char);

vector signed int vec_any_gt (vector unsigned char,
```

```
                              vector unsigned char);
vector signed int vec_any_gt (vector signed char, vector signed char);
vector signed int vec_any_gt (vector signed short,
                              vector unsigned short);
vector signed int vec_any_gt (vector unsigned short,
                              vector signed short);
vector signed int vec_any_gt (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_gt (vector signed short, vector signed short);

vector signed int vec_any_gt (vector signed int, vector unsigned int);
vector signed int vec_any_gt (vector unsigned int, vector signed int);
vector signed int vec_any_gt (vector unsigned int, vector unsigned int);

vector signed int vec_any_gt (vector signed int, vector signed int);
vector signed int vec_any_gt (vector float, vector float);

vector signed int vec_any_le (vector signed char, vector unsigned char);

vector signed int vec_any_le (vector unsigned char, vector signed char);

vector signed int vec_any_le (vector unsigned char,
                              vector unsigned char);
vector signed int vec_any_le (vector signed char, vector signed char);
vector signed int vec_any_le (vector signed short,
                              vector unsigned short);
vector signed int vec_any_le (vector unsigned short,
                              vector signed short);
vector signed int vec_any_le (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_le (vector signed short, vector signed short);

vector signed int vec_any_le (vector signed int, vector unsigned int);
vector signed int vec_any_le (vector unsigned int, vector signed int);
vector signed int vec_any_le (vector unsigned int, vector unsigned int);

vector signed int vec_any_le (vector signed int, vector signed int);
vector signed int vec_any_le (vector float, vector float);

vector signed int vec_any_lt (vector signed char, vector unsigned char);

vector signed int vec_any_lt (vector unsigned char, vector signed char);

vector signed int vec_any_lt (vector unsigned char,
                              vector unsigned char);
vector signed int vec_any_lt (vector signed char, vector signed char);
vector signed int vec_any_lt (vector signed short,
                              vector unsigned short);
vector signed int vec_any_lt (vector unsigned short,
```

```
                              vector signed short);
vector signed int vec_any_lt (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_lt (vector signed short, vector signed short);

vector signed int vec_any_lt (vector signed int, vector unsigned int);
vector signed int vec_any_lt (vector unsigned int, vector signed int);
vector signed int vec_any_lt (vector unsigned int, vector unsigned int);

vector signed int vec_any_lt (vector signed int, vector signed int);
vector signed int vec_any_lt (vector float, vector float);

vector signed int vec_any_nan (vector float);

vector signed int vec_any_ne (vector signed char, vector unsigned char);

vector signed int vec_any_ne (vector signed char, vector signed char);
vector signed int vec_any_ne (vector unsigned char, vector signed char);

vector signed int vec_any_ne (vector unsigned char,
                              vector unsigned char);
vector signed int vec_any_ne (vector signed short,
                              vector unsigned short);
vector signed int vec_any_ne (vector signed short, vector signed short);

vector signed int vec_any_ne (vector unsigned short,
                              vector signed short);
vector signed int vec_any_ne (vector unsigned short,
                              vector unsigned short);
vector signed int vec_any_ne (vector signed int, vector unsigned int);
vector signed int vec_any_ne (vector signed int, vector signed int);
vector signed int vec_any_ne (vector unsigned int, vector signed int);
vector signed int vec_any_ne (vector unsigned int, vector unsigned int);

vector signed int vec_any_ne (vector float, vector float);

vector signed int vec_any_nge (vector float, vector float);

vector signed int vec_any_ngt (vector float, vector float);

vector signed int vec_any_nle (vector float, vector float);

vector signed int vec_any_nlt (vector float, vector float);

vector signed int vec_any_numeric (vector float);

vector signed int vec_any_out (vector float, vector float);
```

5.47 Pragmas Accepted by GCC

GCC supports several types of pragmas, primarily in order to compile code originally written for other compilers. Note that in general we do not recommend the use of pragmas; See Section 5.26 [Function Attributes], page 172, for further explanation.

5.47.1 ARM Pragmas

The ARM target defines pragmas for controlling the default addition of `long_call` and `short_call` attributes to functions. See Section 5.26 [Function Attributes], page 172, for information about the effects of these attributes.

`long_calls`
: Set all subsequent functions to have the `long_call` attribute.

`no_long_calls`
: Set all subsequent functions to have the `short_call` attribute.

`long_calls_off`
: Do not affect the `long_call` or `short_call` attributes of subsequent functions.

5.47.2 Darwin Pragmas

The following pragmas are available for all architectures running the Darwin operating system. These are useful for compatibility with other MacOS compilers.

`mark` *tokens*...
: This pragma is accepted, but has no effect.

`options align=`*alignment*
: This pragma sets the alignment of fields in structures. The values of *alignment* may be `mac68k`, to emulate m68k alignment, or `power`, to emulate PowerPC alignment. Uses of this pragma nest properly; to restore the previous setting, use `reset` for the *alignment*.

`segment` *tokens*...
: This pragma is accepted, but has no effect.

`unused (`*var* `[,` *var*`]...)`
: This pragma declares variables to be possibly unused. GCC will not produce warnings for the listed variables. The effect is similar to that of the `unused` attribute, except that this pragma may appear anywhere within the variables' scopes.

5.47.3 Solaris Pragmas

For compatibility with the SunPRO compiler, the following pragma is supported.

`redefine_extname` *oldname newname*
: This pragma gives the C function *oldname* the assembler label *newname*. The pragma must appear before the function declaration. This pragma is equivalent to the asm labels extension (see Section 5.38 [Asm Labels], page 210). The preprocessor defines `__PRAGMA_REDEFINE_EXTNAME` if the pragma is available.

5.47.4 Tru64 Pragmas

For compatibility with the Compaq C compiler, the following pragma is supported.

`extern_prefix` *string*

This pragma renames all subsequent function and variable declarations such that *string* is prepended to the name. This effect may be terminated by using another `extern_prefix` pragma with the empty string.

This pragma is similar in intent to to the asm labels extension (see Section 5.38 [Asm Labels], page 210) in that the system programmer wants to change the assembly-level ABI without changing the source-level API. The preprocessor defines `__EXTERN_PREFIX` if the pragma is available.

5.48 Unnamed struct/union fields within structs/unions.

For compatibility with other compilers, GCC allows you to define a structure or union that contains, as fields, structures and unions without names. For example:

```
struct {
  int a;
  union {
    int b;
    float c;
  };
  int d;
} foo;
```

In this example, the user would be able to access members of the unnamed union with code like '`foo.b`'. Note that only unnamed structs and unions are allowed, you may not have, for example, an unnamed `int`.

You must never create such structures that cause ambiguous field definitions. For example, this structure:

```
struct {
  int a;
  struct {
    int a;
  };
} foo;
```

It is ambiguous which `a` is being referred to with '`foo.a`'. Such constructs are not supported and must be avoided. In the future, such constructs may be detected and treated as compilation errors.

6 Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro `__GNUC__`. You can also use `__GNUG__` to test specifically for GNU C++ (see section “Standard Predefined Macros” in *The C Preprocessor*).

6.1 Minimum and Maximum Operators in C++

It is very convenient to have operators which return the “minimum” or the “maximum” of two arguments. In GNU C++ (but not in GNU C),

a `<?` *b* is the *minimum*, returning the smaller of the numeric values *a* and *b*;

a `>?` *b* is the *maximum*, returning the larger of the numeric values *a* and *b*.

These operations are not primitive in ordinary C++, since you can use a macro to return the minimum of two things in C++, as in the following example.

```
#define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

You might then use ‘`int min = MIN (i, j);`’ to set *min* to the minimum value of variables *i* and *j*.

However, side effects in `X` or `Y` may cause unintended behavior. For example, `MIN (i++, j++)` will fail, incrementing the smaller counter twice. A GNU C extension allows you to write safe macros that avoid this kind of problem (see Section 5.6 [Naming an Expression’s Type], page 160). However, writing `MIN` and `MAX` as macros also forces you to use function-call notation for a fundamental arithmetic operation. Using GNU C++ extensions, you can write ‘`int min = i <? j;`’ instead.

Since `<?` and `>?` are built into the compiler, they properly handle expressions with side-effects; ‘`int min = i++ <? j++;`’ works correctly.

6.2 When is a Volatile Object Accessed?

Both the C and C++ standard have the concept of volatile objects. These are normally accessed by pointers and used for accessing hardware. The standards encourage compilers to refrain from optimizations concerning accesses to volatile objects that it might perform on non-volatile objects. The C standard leaves it implementation defined as to what constitutes a volatile access. The C++ standard omits to specify this, except to say that C++ should behave in a similar manner to C with respect to volatiles, where possible. The minimum either standard specifies is that at a sequence point all previous accesses to volatile objects have stabilized and no subsequent accesses have occurred. Thus an implementation is free to reorder and combine volatile accesses which occur between sequence points, but cannot do so for accesses across a sequence point. The use of volatiles does not allow you to violate the restriction on updating objects multiple times within a sequence point.

In most expressions, it is intuitively obvious what is a read and what is a write. For instance

```
volatile int *dst = somevalue;
volatile int *src = someothervalue;
*dst = *src;
```

will cause a read of the volatile object pointed to by *src* and stores the value into the volatile object pointed to by *dst*. There is no guarantee that these reads and writes are atomic, especially for objects larger than `int`.

Less obvious expressions are where something which looks like an access is used in a void context. An example would be,

```
volatile int *src = somevalue;
*src;
```

With C, such expressions are rvalues, and as rvalues cause a read of the object, GCC interprets this as a read of the volatile being pointed to. The C++ standard specifies that such expressions do not undergo lvalue to rvalue conversion, and that the type of the dereferenced object may be incomplete. The C++ standard does not specify explicitly that it is this lvalue to rvalue conversion which is responsible for causing an access. However, there is reason to believe that it is, because otherwise certain simple expressions become undefined. However, because it would surprise most programmers, G++ treats dereferencing a pointer to volatile object of complete type in a void context as a read of the object. When the object has incomplete type, G++ issues a warning.

```
struct S;
struct T {int m;};
volatile S *ptr1 = somevalue;
volatile T *ptr2 = somevalue;
*ptr1;
*ptr2;
```

In this example, a warning is issued for `*ptr1`, and `*ptr2` causes a read of the object pointed to. If you wish to force an error on the first case, you must force a conversion to rvalue with, for instance a static cast, `static_cast<S>(*ptr1)`.

When using a reference to volatile, G++ does not treat equivalent expressions as accesses to volatiles, but instead issues a warning that no volatile is accessed. The rationale for this is that otherwise it becomes difficult to determine where volatile access occur, and not possible to ignore the return value from functions returning volatile references. Again, if you wish to force a read, cast the reference to an rvalue.

6.3 Restricting Pointer Aliasing

As with gcc, g++ understands the C99 feature of restricted pointers, specified with the `__restrict__`, or `__restrict` type qualifier. Because you cannot compile C++ by specifying the '`-std=c99`' language flag, `restrict` is not a keyword in C++.

In addition to allowing restricted pointers, you can specify restricted references, which indicate that the reference is not aliased in the local context.

```
void fn (int *__restrict__ rptr, int &__restrict__ rref)
{
  ...
}
```

In the body of `fn`, *rptr* points to an unaliased integer and *rref* refers to a (different) unaliased integer.

You may also specify whether a member function's *this* pointer is unaliased by using `__restrict__` as a member function qualifier.

```
void T::fn () __restrict__
{
  ...
}
```

Within the body of `T::fn`, *this* will have the effective definition `T *__restrict__ const this`. Notice that the interpretation of a `__restrict__` member function qualifier is different to that of `const` or `volatile` qualifier, in that it is applied to the pointer rather than the object. This is consistent with other compilers which implement restricted pointers.

As with all outermost parameter qualifiers, `__restrict__` is ignored in function definition matching. This means you only need to specify `__restrict__` in a function definition, rather than in a function prototype as well.

6.4 Vague Linkage

There are several constructs in C++ which require space in the object file but are not clearly tied to a single translation unit. We say that these constructs have "vague linkage". Typically such constructs are emitted wherever they are needed, though sometimes we can be more clever.

Inline Functions

Inline functions are typically defined in a header file which can be included in many different compilations. Hopefully they can usually be inlined, but sometimes an out-of-line copy is necessary, if the address of the function is taken or if inlining fails. In general, we emit an out-of-line copy in all translation units where one is needed. As an exception, we only emit inline virtual functions with the vtable, since it will always require a copy.

Local static variables and string constants used in an inline function are also considered to have vague linkage, since they must be shared between all inlined and out-of-line instances of the function.

VTables

C++ virtual functions are implemented in most compilers using a lookup table, known as a vtable. The vtable contains pointers to the virtual functions provided by a class, and each object of the class contains a pointer to its vtable (or vtables, in some multiple-inheritance situations). If the class declares any non-inline, non-pure virtual functions, the first one is chosen as the "key method" for the class, and the vtable is only emitted in the translation unit where the key method is defined.

Note: If the chosen key method is later defined as inline, the vtable will still be emitted in every translation unit which defines it. Make sure that any inline virtuals are declared inline in the class body, even if they are not defined there.

type_info objects

C++ requires information about types to be written out in order to implement '`dynamic_cast`', '`typeid`' and exception handling. For polymorphic classes

(classes with virtual functions), the type_info object is written out along with the vtable so that '`dynamic_cast`' can determine the dynamic type of a class object at runtime. For all other types, we write out the type_info object when it is used: when applying '`typeid`' to an expression, throwing an object, or referring to a type in a catch clause or exception specification.

Template Instantiations
: Most everything in this section also applies to template instantiations, but there are other options as well. See Section 6.6 [Where's the Template?], page 254.

When used with GNU ld version 2.8 or later on an ELF system such as Linux/GNU or Solaris 2, or on Microsoft Windows, duplicate copies of these constructs will be discarded at link time. This is known as COMDAT support.

On targets that don't support COMDAT, but do support weak symbols, GCC will use them. This way one copy will override all the others, but the unused copies will still take up space in the executable.

For targets which do not support either COMDAT or weak symbols, most entities with vague linkage will be emitted as local symbols to avoid duplicate definition errors from the linker. This will not happen for local statics in inlines, however, as having multiple copies will almost certainly break things.

See Section 6.5 [Declarations and Definitions in One Header], page 252, for another way to control placement of these constructs.

6.5 Declarations and Definitions in One Header

C++ object definitions can be quite complex. In principle, your source code will need two kinds of things for each object that you use across more than one source file. First, you need an *interface* specification, describing its structure with type declarations and function prototypes. Second, you need the *implementation* itself. It can be tedious to maintain a separate interface description in a header file, in parallel to the actual implementation. It is also dangerous, since separate interface and implementation definitions may not remain parallel.

With GNU C++, you can use a single header file for both purposes.

> *Warning:* The mechanism to specify this is in transition. For the nonce, you must use one of two `#pragma` commands; in a future release of GNU C++, an alternative mechanism will make these `#pragma` commands unnecessary.

The header file contains the full definitions, but is marked with '`#pragma interface`' in the source code. This allows the compiler to use the header file only as an interface specification when ordinary source files incorporate it with `#include`. In the single source file where the full implementation belongs, you can use either a naming convention or '`#pragma implementation`' to indicate this alternate use of the header file.

`#pragma interface`
`#pragma interface "`*subdir/objects*`.h"`
: Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information,

and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing '`#pragma interface`' is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses '`#pragma implementation`'). Instead, the object files will contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to '`#pragma implementation`'.

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

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use '`#pragma interface`'. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use '`#pragma implementation`' with no argument, it applies to an include file with the same basename[1] as your source file. For example, in '`allclass.cc`', giving just '`#pragma implementation`' by itself is equivalent to '`#pragma implementation "allclass.h"`'.

In versions of GNU C++ prior to 2.6.0 '`allclass.h`' was treated as an implementation file whenever you would include it from '`allclass.cc`' even if you never specified '`#pragma implementation`'. This was deemed to be more trouble than it was worth, however, and disabled.

If you use an explicit '`#pragma implementation`', it must appear in your source file *before* you include the affected header files.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use '`#include`' to include the header file; '`#pragma implementation`' only specifies how to use the file—it doesn't actually include it.)

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

'`#pragma implementation`' and '`#pragma interface`' also have an effect on function inlining.

If you define a class in a header file marked with '`#pragma interface`', the effect on a function defined in that class is similar to an explicit `extern` declaration—the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as '`#pragma implementation`', the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without

[1] A file's *basename* was the name stripped of all leading path information and of trailing suffixes, such as '`.h`' or '`.C`' or '`.cc`'.

inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with '`-fno-implement-inlines`'. If any calls were not inlined, you will get linker errors.

6.6 Where's the Template?

C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system. Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which I will refer to as the Borland model and the Cfront model.

Borland model

Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; the compiler emits template instances in each translation unit that uses them, and the linker collapses them together. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. This disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all templates in the header file, since they must be seen to be instantiated.

Cfront model

The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. A more modern version of the repository works as follows: As individual object files are built, the compiler places any template definitions and instantiations encountered in the repository. At link time, the link wrapper adds in the objects in the repository and compiles any needed instances that were not previously emitted. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; for some code this can be just as transparent, but in practice it can been very difficult to build multiple programs in one directory and one program in multiple directories. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which should be compiled separately.

When used with GNU ld version 2.8 or later on an ELF system such as Linux/GNU or Solaris 2, or on Microsoft Windows, g++ supports the Borland model. On other systems, g++ implements neither automatic model.

A future version of g++ will support a hybrid model whereby the compiler will emit any instantiations for which the template definition is included in the compile, and store template definitions and instantiation context information into the object file for the rest. The link wrapper will extract that information as necessary and invoke the compiler to produce the remaining instantiations. The linker will then combine duplicate instantiations.

In the mean time, you have the following options for dealing with template instantiations:

1. Compile your template-using code with '-frepo'. The compiler will generate files with the extension '.rpo' listing all of the template instantiations used in the corresponding object files which could be instantiated there; the link wrapper, 'collect2', will then update the '.rpo' files to tell the compiler where to place those instantiations and rebuild any affected object files. The link-time overhead is negligible after the first pass, as the compiler will continue to place the instantiations in the same files.

   This is your best option for application code written for the Borland model, as it will just work. Code written for the Cfront model will need to be modified so that the template definitions are available at one or more points of instantiation; usually this is as simple as adding `#include <tmethods.cc>` to the end of each template header.

   For library code, if you want the library to provide all of the template instantiations it needs, just try to link all of its object files together; the link will fail, but cause the instantiations to be generated as a side effect. Be warned, however, that this may cause conflicts if multiple libraries try to provide the same instantiations. For greater control, use explicit instantiation as described in the next option.

2. Compile your code with '-fno-implicit-templates' to disable the implicit generation of template instances, and explicitly instantiate all the ones you use. This approach requires more knowledge of exactly which instances you need than do the others, but it's less mysterious and allows greater control. You can scatter the explicit instantiations throughout your program, perhaps putting them in the translation units where the instances are used or the translation units that define the templates themselves; you can put all of the explicit instantiations you need into one big file; or you can create small files like

   ```
   #include "Foo.h"
   #include "Foo.cc"

   template class Foo<int>;
   template ostream& operator <<
                   (ostream&, const Foo<int>&);
   ```

   for each of the instances you need, and create a template instantiation library from those.

   If you are using Cfront-model code, you can probably get away with not using '-fno-implicit-templates' when compiling files that don't '#include' the member template definitions.

   If you use one big file to do the instantiations, you may want to compile it without '-fno-implicit-templates' so you get all of the instances required by your explicit instantiations (but not by any other files) without having to specify them as well.

   g++ has extended the template instantiation syntax outlined in the Working Paper to allow forward declaration of explicit instantiations (with `extern`), instantiation of the compiler support data for a template class (i.e. the vtable) without instantiating any of its members (with `inline`), and instantiation of only the static data members of a template class, without the support data or member functions (with (`static`):

   ```
   extern template int max (int, int);
   inline template class Foo<int>;
   ```

```
static template class Foo<int>;
```

3. Do nothing. Pretend g++ does implement automatic instantiation management. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates it uses. In a large program, this can lead to an unacceptable amount of code duplication.
4. Add '`#pragma interface`' to all files containing template definitions. For each of these files, add '`#pragma implementation "`*filename*`"`' to the top of some '`.C`' file which '`#include`'s it. Then compile everything with '`-fexternal-templates`'. The templates will then only be expanded in the translation unit which implements them (i.e. has a '`#pragma implementation`' line for the file where they live); all other files will use external references. If you're lucky, everything should work properly. If you get undefined symbol errors, you need to make sure that each template instance which is used in the program is used in the file which implements that template. If you don't have any use for a particular instance in that file, you can just instantiate it explicitly, using the syntax from the latest C++ working paper:

   ```
   template class A<int>;
   template ostream& operator << (ostream&, const A<int>&);
   ```

   This strategy will work with code written for either model. If you are using code written for the Cfront model, the file containing a class template and the file containing its member templates should be implemented in the same translation unit.
5. A slight variation on this approach is to use the flag '`-falt-external-templates`' instead. This flag causes template instances to be emitted in the translation unit that implements the header where they are first instantiated, rather than the one which implements the file where the templates are defined. This header must be the same in all translation units, or things are likely to break.

   See Section 6.5 [Declarations and Definitions in One Header], page 252, for more discussion of these pragmas.

6.7 Extracting the function pointer from a bound pointer to member function

In C++, pointer to member functions (PMFs) are implemented using a wide pointer of sorts to handle all the possible call mechanisms; the PMF needs to store information about how to adjust the '`this`' pointer, and if the function pointed to is virtual, where to find the vtable, and where in the vtable to look for the member function. If you are using PMFs in an inner loop, you should really reconsider that decision. If that is not an option, you can extract the pointer to the function that would be called for a given object/PMF pair and call it directly inside the inner loop, to save a bit of time.

Note that you will still be paying the penalty for the call through a function pointer; on most modern architectures, such a call defeats the branch prediction features of the CPU. This is also true of normal virtual function calls.

The syntax for this extension is

```
extern A a;
extern int (A::*fp)();
typedef int (*fptr)(A *);
```

```
fptr p = (fptr)(a.*fp);
```

For PMF constants (i.e. expressions of the form '&Klasse::Member'), no object is needed to obtain the address of the function. They can be converted to function pointers directly:

```
fptr p1 = (fptr)(&A::foo);
```

You must specify '-Wno-pmf-conversions' to use this extension.

6.8 C++-Specific Variable, Function, and Type Attributes

Some attributes only make sense for C++ programs.

`init_priority (`*priority*`)`
: In Standard C++, objects defined at namespace scope are guaranteed to be initialized in an order in strict accordance with that of their definitions *in a given translation unit*. No guarantee is made for initializations across translation units. However, GNU C++ allows users to control the order of initialization of objects defined at namespace scope with the `init_priority` attribute by specifying a relative *priority*, a constant integral expression currently bounded between 101 and 65535 inclusive. Lower numbers indicate a higher priority.

  In the following example, `A` would normally be created before `B`, but the `init_priority` attribute has reversed that order:

  ```
  Some_Class  A  __attribute__ ((init_priority (2000)));
  Some_Class  B  __attribute__ ((init_priority (543)));
  ```

  Note that the particular values of *priority* do not matter; only their relative ordering.

`java_interface`
: This type attribute informs C++ that the class is a Java interface. It may only be applied to classes declared within an `extern "Java"` block. Calls to methods declared in this interface will be dispatched using GCJ's interface table mechanism, instead of regular virtual table dispatch.

6.9 Java Exceptions

The Java language uses a slightly different exception handling model from C++. Normally, GNU C++ will automatically detect when you are writing C++ code that uses Java exceptions, and handle them appropriately. However, if C++ code only needs to execute destructors when Java exceptions are thrown through it, GCC will guess incorrectly. Sample problematic code is:

```
struct S { ~S(); };
extern void bar();    // is written in Java, and may throw exceptions
void foo()
{
  S s;
  bar();
}
```

The usual effect of an incorrect guess is a link failure, complaining of a missing routine called '`__gxx_personality_v0`'.

You can inform the compiler that Java exceptions are to be used in a translation unit, irrespective of what it might think, by writing '`#pragma GCC java_exceptions`' at the head of the file. This '`#pragma`' must appear before any functions that throw or catch exceptions, or run destructors when exceptions are thrown through them.

You cannot mix Java and C++ exceptions in the same translation unit. It is believed to be safe to throw a C++ exception from one file through another file compiled for the Java exception model, or vice versa, but there may be bugs in this area.

6.10 Deprecated Features

In the past, the GNU C++ compiler was extended to experiment with new features, at a time when the C++ language was still evolving. Now that the C++ standard is complete, some of those features are superseded by superior alternatives. Using the old features might cause a warning in some cases that the feature will be dropped in the future. In other cases, the feature might be gone already.

While the list below is not exhaustive, it documents some of the options that are now deprecated:

`-fexternal-templates`
`-falt-external-templates`
: These are two of the many ways for g++ to implement template instantiation. See Section 6.6 [Template Instantiation], page 254. The C++ standard clearly defines how template definitions have to be organized across implementation units. g++ has an implicit instantiation mechanism that should work just fine for standard-conforming code.

`-fstrict-prototype`
`-fno-strict-prototype`
: Previously it was possible to use an empty prototype parameter list to indicate an unspecified number of parameters (like C), rather than no parameters, as C++ demands. This feature has been removed, except where it is required for backwards compatibility See Section 6.11 [Backwards Compatibility], page 259.

The named return value extension has been deprecated, and is now removed from g++.

The use of initializer lists with new expressions has been deprecated, and is now removed from g++.

Floating and complex non-type template parameters have been deprecated, and are now removed from g++.

The implicit typename extension has been deprecated and will be removed from g++ at some point. In some cases g++ determines that a dependant type such as `TPL<T>::X` is a type without needing a `typename` keyword, contrary to the standard.

6.11 Backwards Compatibility

Now that there is a definitive ISO standard C++, G++ has a specification to adhere to. The C++ language evolved over time, and features that used to be acceptable in previous drafts of the standard, such as the ARM [Annotated C++ Reference Manual], are no longer accepted. In order to allow compilation of C++ written to such drafts, G++ contains some backwards compatibilities. *All such backwards compatibility features are liable to disappear in future versions of G++.* They should be considered deprecated See Section 6.10 [Deprecated Features], page 258.

`For scope`
: If a variable is declared at for scope, it used to remain in scope until the end of the scope which contained the for statement (rather than just within the for scope). G++ retains this, but issues a warning, if such a variable is accessed outside the for scope.

`Implicit C language`
: Old C system header files did not contain an `extern "C" {...}` scope to set the language. On such systems, all header files are implicitly scoped inside a C language scope. Also, an empty prototype `()` will be treated as an unspecified number of arguments, rather than no arguments, as C++ demands.

7 GNU Objective-C runtime features

This document is meant to describe some of the GNU Objective-C runtime features. It is not intended to teach you Objective-C, there are several resources on the Internet that present the language. Questions and comments about this document to Ovidiu Predescu `ovidiu@cup.hp.com`.

7.1 `+load`: Executing code before main

The GNU Objective-C runtime provides a way that allows you to execute code before the execution of the program enters the `main` function. The code is executed on a per-class and a per-category basis, through a special class method `+load`.

This facility is very useful if you want to initialize global variables which can be accessed by the program directly, without sending a message to the class first. The usual way to initialize global variables, in the `+initialize` method, might not be useful because `+initialize` is only called when the first message is sent to a class object, which in some cases could be too late.

Suppose for example you have a `FileStream` class that declares `Stdin`, `Stdout` and `Stderr` as global variables, like below:

```
FileStream *Stdin = nil;
FileStream *Stdout = nil;
FileStream *Stderr = nil;

@implementation FileStream

+ (void)initialize
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

In this example, the initialization of `Stdin`, `Stdout` and `Stderr` in `+initialize` occurs too late. The programmer can send a message to one of these objects before the variables are actually initialized, thus sending messages to the `nil` object. The `+initialize` method which actually initializes the global variables is not invoked until the first message is sent to the class object. The solution would require these variables to be initialized just before entering `main`.

The correct solution of the above problem is to use the `+load` method instead of `+initialize`:

```
@implementation FileStream
```

```
+ (void)load
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

The `+load` is a method that is not overridden by categories. If a class and a category of it both implement `+load`, both methods are invoked. This allows some additional initializations to be performed in a category.

This mechanism is not intended to be a replacement for `+initialize`. You should be aware of its limitations when you decide to use it instead of `+initialize`.

7.1.1 What you can and what you cannot do in `+load`

The `+load` implementation in the GNU runtime guarantees you the following things:

- you can write whatever C code you like;
- you can send messages to Objective-C constant strings (`@"this is a constant string"`);
- you can allocate and send messages to objects whose class is implemented in the same file;
- the `+load` implementation of all super classes of a class are executed before the `+load` of that class is executed;
- the `+load` implementation of a class is executed before the `+load` implementation of any category.

In particular, the following things, even if they can work in a particular case, are not guaranteed:

- allocation of or sending messages to arbitrary objects;
- allocation of or sending messages to objects whose classes have a category implemented in the same file;

You should make no assumptions about receiving `+load` in sibling classes when you write `+load` of a class. The order in which sibling classes receive `+load` is not guaranteed.

The order in which `+load` and `+initialize` are called could be problematic if this matters. If you don't allocate objects inside `+load`, it is guaranteed that `+load` is called before `+initialize`. If you create an object inside `+load` the `+initialize` method of object's class is invoked even if `+load` was not invoked. Note if you explicitly call `+load` on a class, `+initialize` will be called first. To avoid possible problems try to implement only one of these methods.

The `+load` method is also invoked when a bundle is dynamically loaded into your running program. This happens automatically without any intervening operation from you. When you write bundles and you need to write `+load` you can safely create and send messages to objects whose classes already exist in the running program. The same restrictions as above apply to classes defined in bundle.

7.2 Type encoding

The Objective-C compiler generates type encodings for all the types. These type encodings are used at runtime to find out information about selectors and methods and about objects and classes.

The types are encoded in the following way:

`char`	`c`
`unsigned char`	`C`
`short`	`s`
`unsigned short`	`S`
`int`	`i`
`unsigned int`	`I`
`long`	`l`
`unsigned long`	`L`
`long long`	`q`
`unsigned long long`	`Q`
`float`	`f`
`double`	`d`
`void`	`v`
`id`	`@`
`Class`	`#`
`SEL`	`:`
`char*`	`*`
unknown type	`?`
bit-fields	`b` followed by the starting position of the bit-field, the type of the bit-field and the size of the bit-field (the bit-fields encoding was changed from the NeXT's compiler encoding, see below)

The encoding of bit-fields has changed to allow bit-fields to be properly handled by the runtime functions that compute sizes and alignments of types that contain bit-fields. The previous encoding contained only the size of the bit-field. Using only this information it is not possible to reliably compute the size occupied by the bit-field. This is very important in the presence of the Boehm's garbage collector because the objects are allocated using the typed memory facility available in this collector. The typed memory allocation requires information about where the pointers are located inside the object.

The position in the bit-field is the position, counting in bits, of the bit closest to the beginning of the structure.

The non-atomic types are encoded as follows:

pointers	'`^`' followed by the pointed type.
arrays	'`[`' followed by the number of elements in the array followed by the type of the elements followed by '`]`'
structures	'`{`' followed by the name of the structure (or '`?`' if the structure is unnamed), the '`=`' sign, the type of the members and by '`}`'

unions — '(' followed by the name of the structure (or '?' if the union is unnamed), the '=' sign, the type of the members followed by ')'

Here are some types and their encodings, as they are generated by the compiler on an i386 machine:

Objective-C type	Compiler encoding
`int a[10];`	`[10i]`
`struct { int i; float f[3]; int a:3; int b:2; char c; }`	`{?=i[3f]b128i3b131i2c}`

```
struct {
  int i;
  float f[3];
  int a:3;
  int b:2;
  char c;
}
```

In addition to the types the compiler also encodes the type specifiers. The table below describes the encoding of the current Objective-C type specifiers:

Specifier	Encoding
`const`	`r`
`in`	`n`
`inout`	`N`
`out`	`o`
`bycopy`	`O`
`oneway`	`V`

The type specifiers are encoded just before the type. Unlike types however, the type specifiers are only encoded when they appear in method argument types.

7.3 Garbage Collection

Support for a new memory management policy has been added by using a powerful conservative garbage collector, known as the Boehm-Demers-Weiser conservative garbage collector. It is available from `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

To enable the support for it you have to configure the compiler using an additional argument, '`--enable-objc-gc`'. You need to have garbage collector installed before building the compiler. This will build an additional runtime library which has several enhancements to support the garbage collector. The new library has a new name, '`libobjc_gc.a`' to not conflict with the non-garbage-collected library.

When the garbage collector is used, the objects are allocated using the so-called typed memory allocation mechanism available in the Boehm-Demers-Weiser collector. This mode requires precise information on where pointers are located inside objects. This information is computed once per class, immediately after the class has been initialized.

There is a new runtime function `class_ivar_set_gcinvisible()` which can be used to declare a so-called *weak pointer* reference. Such a pointer is basically hidden for the garbage collector; this can be useful in certain situations, especially when you want to keep track of the allocated objects, yet allow them to be collected. This kind of pointers can only be members of objects, you cannot declare a global pointer as a weak reference. Every type which is a pointer type can be declared a weak pointer, including `id`, `Class` and `SEL`.

Here is an example of how to use this feature. Suppose you want to implement a class whose instances hold a weak pointer reference; the following class does this:

```
@interface WeakPointer : Object
{
    const void* weakPointer;
}

- initWithPointer:(const void*)p;
- (const void*)weakPointer;
@end


@implementation WeakPointer

+ (void)initialize
{
  class_ivar_set_gcinvisible (self, "weakPointer", YES);
}

- initWithPointer:(const void*)p
{
  weakPointer = p;
  return self;
}

- (const void*)weakPointer
{
  return weakPointer;
}

@end
```

Weak pointers are supported through a new type character specifier represented by the '`!`' character. The `class_ivar_set_gcinvisible()` function adds or removes this specifier to the string type description of the instance variable named as argument.

7.4 Constant string objects

GNU Objective-C provides constant string objects that are generated directly by the compiler. You declare a constant string object by prefixing a C constant string with the character '`@`':

```
id myString = @"this is a constant string object";
```

The constant string objects are usually instances of the `NXConstantString` class which is provided by the GNU Objective-C runtime. To get the definition of this class you must include the '`objc/NXConstStr.h`' header file.

User defined libraries may want to implement their own constant string class. To be able to support them, the GNU Objective-C compiler provides a new command line options '`-fconstant-string-class=`*class-name*'. The provided class should adhere to a strict structure, the same as `NXConstantString`'s structure:

```
@interface NXConstantString : Object
{
  char *c_string;
  unsigned int len;
}
@end
```

User class libraries may choose to inherit the customized constant string class from a different class than `Object`. There is no requirement in the methods the constant string class has to implement.

When a file is compiled with the '`-fconstant-string-class`' option, all the constant string objects will be instances of the class specified as argument to this option. It is possible to have multiple compilation units referring to different constant string classes, neither the compiler nor the linker impose any restrictions in doing this.

7.5 compatibility_alias

This is a feature of the Objective-C compiler rather than of the runtime, anyway since it is documented nowhere and its existence was forgotten, we are documenting it here.

The keyword `@compatibility_alias` allows you to define a class name as equivalent to another class name. For example:

```
@compatibility_alias WOApplication GSWApplication;
```

tells the compiler that each time it encounters `WOApplication` as a class name, it should replace it with `GSWApplication` (that is, `WOApplication` is just an alias for `GSWApplication`).

There are some constraints on how this can be used—

- `WOApplication` (the alias) must not be an existing class;
- `GSWApplication` (the real class) must be an existing class.

8 gcov: a Test Coverage Program

`gcov` is a tool you can use in conjunction with GCC to test code coverage in your programs.

8.1 Introduction to gcov

`gcov` is a test coverage program. Use it in concert with GCC to analyze your programs to help create more efficient, faster running code. You can use `gcov` as a profiling tool to help discover where your optimization efforts will best affect your code. You can also use `gcov` along with the other profiling tool, `gprof`, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as `gcov` or `gprof`, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. `gcov` helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use `gcov` because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for 'hot spots' where the code is using a great deal of computer time. Likewise, because `gcov` accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful—they only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

`gcov` creates a logfile called '*sourcefile*`.gcov`' which indicates how many times each line of a source file '*sourcefile*`.c`' has executed. You can use these logfiles along with `gprof` to aid in fine-tuning the performance of your programs. `gprof` gives timing information you can use along with the information you get from `gcov`.

`gcov` works only on code compiled with GCC. It is not compatible with any other profiling or test coverage mechanism.

8.2 Invoking gcov

gcov [*options*] *sourcefile*

gcov accepts the following options:

-h
--help
: Display help about using gcov (on the standard output), and exit without doing any further processing.

-v
--version
: Display the gcov version number (on the standard output), and exit without doing any further processing.

-b
--branch-probabilities
: Write branch frequencies to the output file, and write branch summary info to the standard output. This option allows you to see how often each branch in your program was taken.

-c
--branch-counts
: Write branch frequencies as the number of branches taken, rather than the percentage of branches taken.

-n
--no-output
: Do not create the gcov output file.

-l
--long-file-names
: Create long file names for included source files. For example, if the header file 'x.h' contains code, and was included in the file 'a.c', then running gcov on the file 'a.c' will produce an output file called 'a.c.x.h.gcov' instead of 'x.h.gcov'. This can be useful if 'x.h' is included in multiple source files.

-f
--function-summaries
: Output summaries for each function in addition to the file level summary.

-o *directory*
--object-directory *directory*
: The directory where the object files live. Gcov will search for '.bb', '.bbg', and '.da' files in this directory.

When using `gcov`, you must first compile your program with two special GCC options: '`-fprofile-arcs -ftest-coverage`'. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by gcov. These additional files are placed in the directory where the source code is located.

Running the program will cause profile output to be generated. For each source file compiled with '`-fprofile-arcs`', an accompanying '`.da`' file will be placed in the source directory.

Running `gcov` with your program's source file names as arguments will now produce a listing of the code along with frequency of execution for each line. For example, if your program is called '`tmp.c`', this is what you see when you use the basic `gcov` facility:

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c
 87.50% of 8 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

The file '`tmp.c.gcov`' contains output from `gcov`. Here is a sample:

```
                main()
                {
           1      int i, total;

           1      total = 0;

          11      for (i = 0; i < 10; i++)
          10        total += i;

           1      if (total != 45)
      ######        printf ("Failure\n");
                  else
           1        printf ("Success\n");
           1    }
```

When you use the '`-b`' option, your output looks like this:

```
$ gcov -b tmp.c
 87.50% of 8 source lines executed in file tmp.c
 80.00% of 5 branches executed in file tmp.c
 80.00% of 5 branches taken at least once in file tmp.c
 50.00% of 2 calls executed in file tmp.c
Creating tmp.c.gcov.
```

Here is a sample of a resulting '`tmp.c.gcov`' file:

```
                main()
                {
           1      int i, total;

           1      total = 0;

          11      for (i = 0; i < 10; i++)
```

```
branch 0 taken = 91%
branch 1 taken = 100%
branch 2 taken = 100%
          10        total += i;

           1      if (total != 45)
branch 0 taken = 100%
      ######        printf ("Failure\n");
call 0 never executed
branch 1 never executed
                  else
           1        printf ("Success\n");
call 0 returns = 100%
           1    }
```

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions call `exit` or `longjmp`, and thus may not return every time they are called.

The execution counts are cumulative. If the example program were executed again without removing the '`.da`' file, the count for the number of times each line in the source was executed would be added to the results of the previous run(s). This is potentially useful in several ways. For example, it could be used to accumulate data over a number of program runs as part of a test verification suite, or to provide more accurate long-term information over a large number of program runs.

The data in the '`.da`' files is saved immediately before the program exits. For each source file compiled with '`-fprofile-arcs`', the profiling code first attempts to read in an existing '`.da`' file; if the file doesn't match the executable (differing number of basic block counts) it will ignore the contents of the file. It then adds in the new execution counts and finally writes the data to the file.

8.3 Using `gcov` with GCC Optimization

If you plan to use `gcov` to help optimize your code, you must first compile your program with two special GCC options: '`-fprofile-arcs -ftest-coverage`'. Aside from that, you can use any other GCC options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for `gcov` to calculate separate execution counts for each line because there isn't separate code for each line. Hence the `gcov` output looks like this if you compiled the program with optimization:

```
          100  if (a != b)
          100    c = 1;
          100  else
          100    c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

8.4 Brief description of gcov data files

`gcov` uses three files for doing profiling. The names of these files are derived from the original *source* file by substituting the file suffix with either '`.bb`', '`.bbg`', or '`.da`'. All of these files are placed in the same directory as the source file, and contain data stored in a platform-independent method.

The '`.bb`' and '`.bbg`' files are generated when the source file is compiled with the GCC '`-ftest-coverage`' option. The '`.bb`' file contains a list of source files (including headers), functions within those files, and line numbers corresponding to each basic block in the source file.

The '`.bb`' file format consists of several lists of 4-byte integers which correspond to the line numbers of each basic block in the file. Each list is terminated by a line number of 0. A line number of -1 is used to designate that the source file name (padded to a 4-byte boundary and followed by another -1) follows. In addition, a line number of -2 is used to designate that the name of a function (also padded to a 4-byte boundary and followed by a -2) follows.

The '`.bbg`' file is used to reconstruct the program flow graph for the source file. It contains a list of the program flow arcs (possible branches taken from one basic block to another) for each function which, in combination with the '`.bb`' file, enables gcov to reconstruct the program flow.

In the '`.bbg`' file, the format is:

```
number of basic blocks for function #0 (4-byte number)
total number of arcs for function #0 (4-byte number)
count of arcs in basic block #0 (4-byte number)
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
destination basic block of arc #1 (4-byte number)
flag bits (4-byte number)
...
```

```
destination basic block of arc #N (4-byte number)
flag bits (4-byte number)
count of arcs in basic block #1 (4-byte number)
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
...
```

A -1 (stored as a 4-byte number) is used to separate each function's list of basic blocks, and to verify that the file has been read correctly.

The '`.da`' file is generated when a program containing object files built with the GCC '`-fprofile-arcs`' option is executed. A separate '`.da`' file is created for each source file compiled with this option, and the name of the '`.da`' file is stored as an absolute pathname in the resulting object file. This path name is derived from the source file name by substituting a '`.da`' suffix.

The format of the '`.da`' file is fairly simple. The first 8-byte number is the number of counts in the file, followed by the counts (stored as 8-byte numbers). Each count corresponds to the number of times each arc in the program is executed. The counts are cumulative; each time the program is executed, it attempts to combine the existing '`.da`' files with the new counts for this invocation of the program. It ignores the contents of any '`.da`' files whose number of arcs doesn't correspond to the current program, and merely overwrites them instead.

All three of these files use the functions in '`gcov-io.h`' to store integers; the functions in this header provide a machine-independent mechanism for storing and retrieving data from a stream.

9 Known Causes of Trouble with GCC

This section describes known problems that affect users of GCC. Most of these are not GCC bugs per se—if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

9.1 Actual Bugs We Haven't Fixed Yet

- The `fixincludes` script interacts badly with automounters; if the directory of system header files is automounted, it tends to be unmounted while `fixincludes` is running. This would seem to be a bug in the automounter. We don't know any good way to work around it.
- The `fixproto` script will sometimes add prototypes for the `sigsetjmp` and `siglongjmp` functions that reference the `jmp_buf` type before that type is defined. To work around this, edit the offending file and place the typedef in front of the prototypes.
- When '`-pedantic-errors`' is specified, GCC will incorrectly give an error message when a function name is specified in an expression involving the comma operator.

9.2 Cross-Compiler Problems

You may run into problems with cross compilation on certain machines, for several reasons.

- Cross compilation can run into trouble for certain machines because some target machines' assemblers require floating point numbers to be written as *integer* constants in certain contexts.

  The compiler writes these integer constants by examining the floating point value as an integer and printing that integer, because this is simple to write and independent of the details of the floating point representation. But this does not work if the compiler is running on a different machine with an incompatible floating point format, or even a different byte-ordering.

  In addition, correct constant folding of floating point values requires representing them in the target machine's format. (The C standard does not quite require this, but in practice it is the only way to win.)

  It is now possible to overcome these problems by defining macros such as `REAL_VALUE_TYPE`. But doing so is a substantial amount of work for each target machine. See section "Cross Compilation and Floating Point" in *GNU Compiler Collection (GCC) Internals*.
- At present, the program '`mips-tfile`' which adds debug support to object files on MIPS systems does not work in a cross compile environment.

9.3 Interoperation

This section lists various difficulties encountered in using GCC together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

- G++ does not do name mangling in the same way as other C++ compilers. This means that object files compiled with one compiler cannot be used with another.

  This effect is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation, including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers—but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.
- Older GDB versions sometimes fail to read the output of GCC version 2. If you have trouble, get GDB version 4.4 or later.
- DBX rejects some files produced by GCC, though it accepts similar constructs in output from PCC. Until someone can supply a coherent description of what is valid DBX input and what is not, there is nothing I can do about these problems. You are on your own.
- The GNU assembler (GAS) does not support PIC. To generate PIC code, you must use some other assembler, such as '`/bin/as`'.
- On some BSD systems, including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.
- On some SGI systems, when you use '`-lgl_s`' as an option, it gets translated magically to '`-lgl_s -lX11_s -lc_s`'. Naturally, this does not happen when you use GCC. You must specify all three options explicitly.
- On a Sparc, GCC aligns all values of type `double` on an 8-byte boundary, and it expects every `double` to be so aligned. The Sun compiler usually gives `double` values 8-byte alignment, with one exception: function arguments of type `double` may not be aligned.

  As a result, if a function compiled with Sun CC takes the address of an argument of type `double` and passes this pointer of type `double *` to a function compiled with GCC, dereferencing the pointer may cause a fatal signal.

  One way to solve this problem is to compile your entire program with GCC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function `access_double` instead of directly with '`*`':

  ```
  inline double
  access_double (double *unaligned_ptr)
  {
    union d2i { double d; int i[2]; };

    union d2i *p = (union d2i *) unaligned_ptr;
    union d2i u;

    u.i[0] = p->i[0];
    u.i[1] = p->i[1];
  ```

```
    return u.d;
  }
```

Storing into the pointer can be done likewise with the same union.

- On Solaris, the `malloc` function in the '`libmalloc.a`' library may allocate memory that is only 4 byte aligned. Since GCC on the Sparc assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the '`libmalloc.a`' library.

  The solution is to not use the '`libmalloc.a`' library. Use instead `malloc` and related functions from '`libc.a`'; they do not have this problem.
- Sun forgot to include a static version of '`libdl.a`' with some versions of SunOS (mainly 4.1). This results in undefined symbols when linking static binaries (that is, if you use '`-static`'). If you see undefined symbols `_dlclose`, `_dlsym` or `_dlopen` when linking, compile and link against the file '`mit/util/misc/dlsym.c`' from the MIT version of X windows.
- The 128-bit long double format that the Sparc port supports currently works by using the architecturally defined quad-word floating point instructions. Since there is no hardware that supports these instructions they must be emulated by the operating system. Long doubles do not work in Sun OS versions 4.0.3 and earlier, because the kernel emulator uses an obsolete and incompatible format. Long doubles do not work in Sun OS version 4.1.1 due to a problem in a Sun library. Long doubles do work on Sun OS versions 4.1.2 and higher, but GCC does not enable them by default. Long doubles appear to work in Sun OS 5.x (Solaris 2.x).
- On HP-UX version 9.01 on the HP PA, the HP compiler `cc` does not compile GCC correctly. We do not yet know why. However, GCC compiled on earlier HP-UX versions works properly on HP-UX 9.01 and can compile itself properly on 9.01.
- On the HP PA machine, ADB sometimes fails to work on functions compiled with GCC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GCC doesn't generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.
- Debugging ('`-g`') is not supported on the HP PA machine, unless you use the preliminary GNU tools.
- Taking the address of a label may generate errors from the HP-UX PA assembler. GAS for the PA does not have this problem.
- Using floating point parameters for indirect calls to static functions will not work when using the HP assembler. There simply is no way for GCC to specify what registers hold arguments for static functions when using the HP assembler. GAS for the PA does not have this problem.
- In extremely rare cases involving some very large functions you may receive errors from the HP linker complaining about an out of bounds unconditional branch offset. This used to occur more often in previous versions of GCC, but is now exceptionally rare. If you should run into it, you can work around by making your function smaller.
- GCC compiled code sometimes emits warnings from the HP-UX assembler of the form:

  ```
  (warning) Use of GR3 when
  ```

```
    frame >= 8192 may cause conflict.
```

These warnings are harmless and can be safely ignored.

- On the IBM RS/6000, compiling code of the form

  ```
  extern int foo;

  ... foo ...

  static int foo;
  ```

  will cause the linker to report an undefined symbol `foo`. Although this behavior differs from most other systems, it is not a bug because redefining an `extern` variable as `static` is undefined in ISO C.

- In extremely rare cases involving some very large functions you may receive errors from the AIX Assembler complaining about a displacement that is too large. If you should run into it, you can work around by making your function smaller.
- The '`libstdc++.a`' library in GCC relies on the SVR4 dynamic linker semantics which merges global symbols between libraries and applications, especially necessary for C++ streams functionality. This is not the default behavior of AIX shared libraries and dynamic linking. '`libstdc++.a`' is built on AIX with "runtime-linking" enabled so that symbol merging can occur. To utilize this feature, the application linked with '`libstdc++.a`' must include the '`-Wl,-brtl`' flag on the link line. G++ cannot impose this because this option may interfere with the semantics of the user program and users may not always use '`g++`' to link his or her application. Applications are not required to use the '`-Wl,-brtl`' flag on the link line—the rest of the '`libstdc++.a`' library which is not dependent on the symbol merging semantics will continue to function correctly.
- An application can interpose its own definition of functions for functions invoked by '`libstdc++.a`' with "runtime-linking" enabled on AIX. To accomplish this the application must be linked with "runtime-linking" option and the functions explicitly must be exported by the application ('`-Wl,-brtl,-bE:exportfile`').
- AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ('`.`' vs '`,`' for separating decimal fractions). There have been problems reported where the library linked with GCC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the `LANG` environment variable to '`C`' or '`En_US`'.
- Even if you specify '`-fdollars-in-identifiers`', you cannot successfully use '`$`' in identifiers on the RS/6000 due to a restriction in the IBM assembler. GAS supports these identifiers.
- There is an assembler bug in versions of DG/UX prior to 5.4.2.01 that occurs when the '`fldcr`' instruction is used. GCC uses '`fldcr`' on the 88100 to serialize volatile memory references. Use the option '`-mno-serialize-volatile`' if your version of the assembler has this bug.
- On VMS, GAS versions 1.38.1 and earlier may cause spurious warning messages from the linker. These warning messages complain of mismatched psect attributes. You can ignore them.

- On NewsOS version 3, if you include both of the files '`stddef.h`' and '`sys/types.h`', you get an error because there are two typedefs of `size_t`. You should change '`sys/types.h`' by adding these lines around the definition of `size_t`:

  ```
  #ifndef _SIZE_T
  #define _SIZE_T
  actual-typedef-here
  #endif
  ```

- On the Alliant, the system's own convention for returning structures and unions is unusual, and is not compatible with GCC no matter what options are used.
- On the IBM RT PC, the MetaWare HighC compiler (hc) uses a different convention for structure and union returning. Use the option '`-mhc-struct-return`' to tell GCC to use a convention compatible with it.
- On Ultrix, the Fortran compiler expects registers 2 through 5 to be saved by function calls. However, the C compiler uses conventions compatible with BSD Unix: registers 2 through 5 may be clobbered by function calls.

  GCC uses the same convention as the Ultrix C compiler. You can use these options to produce code compatible with the Fortran compiler:

  ```
  -fcall-saved-r2 -fcall-saved-r3 -fcall-saved-r4 -fcall-saved-r5
  ```

- On the WE32k, you may find that programs compiled with GCC do not work with the standard shared C library. You may need to link with the ordinary C compiler. If you do so, you must specify the following options:

  ```
  -L/usr/local/lib/gcc-lib/we32k-att-sysv/2.8.1 -lgcc -lc_s
  ```

  The first specifies where to find the library '`libgcc.a`' specified with the '`-lgcc`' option.

  GCC does linking by invoking `ld`, just as `cc` does, and there is no reason why it *should* matter which compilation program you use to invoke `ld`. If someone tracks this problem down, it can probably be fixed easily.

- On the Alpha, you may get assembler errors about invalid syntax as a result of floating point constants. This is due to a bug in the C library functions `ecvt`, `fcvt` and `gcvt`. Given valid floating point numbers, they sometimes print '`NaN`'.
- On Irix 4.0.5F (and perhaps in some other versions), an assembler bug sometimes reorders instructions incorrectly when optimization is turned on. If you think this may be happening to you, try using the GNU assembler; GAS version 2.1 supports ECOFF on Irix.

  Or use the '`-noasmopt`' option when you compile GCC with itself, and then again when you compile your program. (This is a temporary kludge to turn off assembler optimization on Irix.) If this proves to be what you need, edit the assembler spec in the file '`specs`' so that it unconditionally passes '`-O0`' to the assembler, and never passes '`-O2`' or '`-O3`'.

9.4 Problems Compiling Certain Programs

Certain programs have problems compiling.

- Parse errors may occur compiling X11 on a Decstation running Ultrix 4.2 because of problems in DEC's versions of the X11 header files '`X11/Xlib.h`' and '`X11/Xutil.h`'.

People recommend adding '-I/usr/include/mit' to use the MIT versions of the header files, using the '-traditional' switch to turn off ISO C, or fixing the header files by adding this:

```
#ifdef __STDC__
#define NeedFunctionPrototypes 0
#endif
```

- On various 386 Unix systems derived from System V, including SCO, ISC, and ESIX, you may get error messages about running out of virtual memory while compiling certain programs.

  You can prevent this problem by linking GCC with the GNU malloc (which thus replaces the malloc that comes with the system). GNU malloc is available as a separate package, and also in the file 'src/gmalloc.c' in the GNU Emacs 19 distribution.

  If you have installed GNU malloc as a separate library package, use this option when you relink GCC:

  ```
  MALLOC=/usr/local/lib/libgmalloc.a
  ```

  Alternatively, if you have compiled 'gmalloc.c' from Emacs 19, copy the object file to 'gmalloc.o' and use this option when you relink GCC:

  ```
  MALLOC=gmalloc.o
  ```

9.5 Incompatibilities of GCC

There are several noteworthy incompatibilities between GNU C and K&R (non-ISO) versions of C. The '-traditional' option eliminates many of these incompatibilities, *but not all*, by telling GCC to behave like a K&R C compiler.

- GCC normally makes string constants read-only. If several identical-looking string constants are used, GCC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '-fwritable-strings' flag, which directs GCC to handle string constants the same way most C compilers do. '-traditional' also has this effect, among others.

- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ISO C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.

- GCC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GCC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument *a* is.

The '-traditional' option directs GCC to handle such cases (among others) in the old-fashioned (non-ISO) fashion.

- When you use setjmp and longjmp, the only automatic variables guaranteed to remain valid are those declared volatile. This is a consequence of automatic register allocation. Consider this function:

```
jmp_buf j;

foo ()
{
  int a, b;

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

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

  Here a may or may not be restored to its first value when the longjmp occurs. If a is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

  If you use the '-W' option with the '-O' option, you will get a warning when GCC thinks such a problem might be possible.

  The '-traditional' option directs GCC to put variables in the stack by default, rather than in registers, in functions that call setjmp. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GCC. For example, a program like this will not work:

```
foobar (
#define luser
        hack)
```

  ISO C does not permit such a construct. It would make sense to support it when '-traditional' is used, but it is too much work to implement.

- K&R compilers allow comments to cross over an inclusion boundary (i.e. started in an include file and ended in the including file). I think this would be quite ugly and can't imagine it could be needed.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a extern declaration affects all the rest of the file even if it happens within a block.

  The '-traditional' option directs GCC 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 ISO C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.
- When in '`-traditional`' mode, GCC allows the following erroneous pair of declarations to appear together in a given scope:

  ```
  typedef int foo;
  typedef foo foo;
  ```
- GCC treats all characters of identifiers as significant, even when in '`-traditional`' mode. According to K&R-1 (2.2), "No more than the first eight characters are significant, although more may be used.". Also according to K&R-1 (2.2), "An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter.", but GCC also allows dollar signs in identifiers.
- PCC allows whitespace in the middle of compound assignment operators such as '`+=`'. GCC, following the ISO standard, does not allow this. The difficulty described immediately above applies here too.
- GCC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GCC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '`/*...*/`'. However, '`-traditional`' suppresses these error messages.
- Many user programs contain the declaration '`long time ();`'. In the past, the system header files on many systems did not actually declare `time`, so it did not matter what type your program declared it to return. But in systems with ISO C headers, `time` is declared to return `time_t`, and if that is not the same as `long`, then '`long time ();`' is erroneous.

  The solution is to change your program to use appropriate system headers (`<time.h>` on systems with ISO C headers) and not to declare `time` if the system header files declare it, or failing that to use `time_t` as the return type of `time`.
- When compiling functions that return `float`, PCC converts it to a double. GCC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.
- When compiling functions that return structures or unions, GCC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GCC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GCC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GCC 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. GCC does not use this method because it is slower and nonreentrant.

On some newer machines, PCC uses a reentrant convention for all structure and union returning. GCC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

You can tell GCC to use a compatible convention for all structure and union returning with the option '`-fpcc-struct-return`'.

- GCC complains about program fragments such as '`0x74ae-0x4000`' which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single *preprocessing token*. Each such token must correspond to one token in C. Since this does not, GCC prints an error message. Although it may appear obvious that what is meant is an operator and two values, the ISO C standard specifically requires that this be treated as erroneous.

  A *preprocessing token* is a *preprocessing number* if it begins with a digit and is followed by letters, underscores, digits, periods and '`e+`', '`e-`', '`E+`', '`E-`', '`p+`', '`p-`', '`P+`', or '`P-`' character sequences. (In strict C89 mode, the sequences '`p+`', '`p-`', '`P+`' and '`P-`' cannot appear in preprocessing numbers.)

  To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

9.6 Fixed Header Files

GCC needs to install corrected versions of some system header files. This is because most target systems have some header files that won't work with GCC unless they are changed. Some have bugs, some are incompatible with ISO C, and some depend on special features of other compilers.

Installing GCC automatically creates and installs the fixed header files, by running a program called `fixincludes` (or for certain targets an alternative such as `fixinc.svr4`). Normally, you don't need to pay attention to this. But there are cases where it doesn't do the right thing automatically.

- If you update the system's header files, such as by installing a new system version, the fixed header files of GCC are not automatically updated. The easiest way to update them is to reinstall GCC. (If you want to be clever, look in the makefile and you can find a shortcut.)
- On some systems, in particular SunOS 4, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of SunOS 4 on different machine models.

The programs that fix the header files do not understand this special way of using symbolic links; therefore, the directory of fixed header files is good only for the machine model used to build it.

In SunOS 4, only programs that look inside the kernel will notice the difference between machine models. Therefore, for most purposes, you need not be concerned about this.

It is possible to make separate sets of fixed header files for the different machine models, and arrange a structure of symbolic links so as to use the proper set, but you'll have to do this by hand.

- On Lynxos, GCC by default does not fix the header files. This is because bugs in the shell cause the `fixincludes` script to fail.

  This means you will encounter problems due to bugs in the system header files. It may be no comfort that they aren't GCC's fault, but it does mean that there's nothing for us to do about them.

9.7 Standard Libraries

GCC by itself attempts to be a conforming freestanding implementation. See Chapter 2 [Language Standards Supported by GCC], page 5, for details of what this means. Beyond the library facilities required of such an implementation, the rest of the C library is supplied by the vendor of the operating system. If that C library doesn't conform to the C standards, then your programs might get warnings (especially when using '`-Wall`') that you don't expect.

For example, the `sprintf` function on SunOS 4.1.3 returns `char *` while the C standard says that `sprintf` returns an `int`. The `fixincludes` program could make the prototype for this function match the Standard, but that would be wrong, since the function will still return `char *`.

If you need a Standard compliant library, then you need to find one, as GCC does not provide one. The GNU C library (called `glibc`) provides ISO C, POSIX, BSD, SystemV and X/Open compatibility for GNU/Linux and HURD-based GNU systems; no recent version of it supports other systems, though some very old versions did. Version 2.2 of the GNU C library includes nearly complete C99 support. You could also ask your operating system vendor if newer libraries are available.

9.8 Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GCC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GCC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- Users often think it is a bug when GCC reports an error for code like this:

  ```
  int foo (struct mumble *);

  struct mumble { ... };

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

  This code really is erroneous, because the scope of `struct mumble` in the prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

  This behavior may seem silly, but it's what the ISO standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ISO C just to avoid an error for the example shown above.

- Accesses to bit-fields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bit-field; it may even vary for a given bit-field according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bit-fields.

- GCC comes with shell scripts to fix certain known problems in system header files. They install corrected copies of various header files in a special directory where only GCC will normally look for them. The scripts adapt to various systems by searching all the system header files for the problem cases that we know about.

  If new system header files are installed, nothing automatically arranges to update the corrected header files. You will have to reinstall GCC to fix the new header files. More specifically, go to the build directory and delete the files '`stmp-fixinc`' and '`stmp-headers`', and the subdirectory `include`; then do '`make install`' again.

- On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the '`-ffloat-store`' option (see Section 3.10 [Optimize Options], page 50).

- On the MIPS, variable argument functions using '`varargs.h`' cannot have a floating point value for the first argument. The reason for this is that in the absence of a prototype in scope, if the first argument is a floating point, it is passed in a floating point register, rather than an integer register.

  If the code is rewritten to use the ISO standard '`stdarg.h`' method of variable arguments, and the prototype is in scope at the time of the call, everything will work fine.

- On the H8/300 and H8/300H, variable argument functions must be implemented using the ISO standard '`stdarg.h`' method of variable arguments. Furthermore, calls to functions using '`stdarg.h`' variable arguments must have a prototype for the called function in scope at the time of the call.
- On AIX and other platforms without weak symbol support, templates need to be instantiated explicitly and symbols for static members of templates will not be generated.

9.9 Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ISO C++ standard) was only recently completed. As a result, your C++ compiler may occasionally surprise you, even when its behavior is correct. This section discusses some areas that frequently give rise to questions of this sort.

9.9.1 Declare *and* Define Static Members

When a class has static data members, it is not enough to *declare* the static member; you must also *define* it. For example:

```
class Foo
{
  ...
  void method();
  static int bar;
};
```

This declaration only establishes that the class `Foo` has an `int` named `Foo::bar`, and a member function named `Foo::method`. But you still need to define *both* `method` and `bar` elsewhere. According to the ISO standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to `g++` from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: `g++` reports as undefined symbols any static data members that lack definitions.

9.9.2 Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like string classes, especially ones that define a conversion function to type `char *` or `const char *`—which is one reason why the standard `string` class requires you to call the `c_str` member function. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function `strfunc` that returns `string` objects, and another function `charfunc` that operates on pointers to `char`:

```
string strfunc ();
void charfunc (const char *);

void
f ()
{
  const char *p = strfunc().c_str();
  ...
  charfunc (p);
  ...
  charfunc (p);
}
```

In this situation, it may seem reasonable to save a pointer to the C string returned by the `c_str` member function and use that rather than call `c_str` repeatedly. However, the temporary string created by the call to `strfunc` is destroyed after `p` is initialized, at which point `p` is left pointing to freed memory.

Code like this may run successfully under some other compilers, particularly obsolete cfront-based compilers that delete temporaries along with normal local variables. However, the GNU C++ behavior is standard-conforming, so if your program depends on late destruction of temporaries it is not portable.

The safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
string& tmp = strfunc ();
charfunc (tmp.c_str ());
```

9.9.3 Implicit Copy-Assignment for Virtual Bases

When a base class is virtual, only one subobject of the base class belongs to each full object. Also, the constructors and destructors are invoked only once, and called from the most-derived class. However, such objects behave unspecified when being assigned. For example:

```
struct Base{
  char *name;
  Base(char *n) : name(strdup(n)){}
  Base& operator= (const Base& other){
   free (name);
   name = strdup (other.name);
  }
};

struct A:virtual Base{
  int val;
  A():Base("A"){}
};

struct B:virtual Base{
  int bval;
```

```
  B():Base("B"){}
};

struct Derived:public A, public B{
  Derived():Base("Derived"){}
};

void func(Derived &d1, Derived &d2)
{
  d1 = d2;
}
```

The C++ standard specifies that 'Base::Base' is only called once when constructing or copy-constructing a Derived object. It is unspecified whether 'Base::operator=' is called more than once when the implicit copy-assignment for Derived objects is invoked (as it is inside 'func' in the example).

g++ implements the "intuitive" algorithm for copy-assignment: assign all direct bases, then assign all members. In that algorithm, the virtual base subobject can be encountered many times. In the example, copying proceeds in the following order: 'val', 'name' (via strdup), 'bval', and 'name' again.

If application code relies on copy-assignment, a user-defined copy-assignment operator removes any uncertainties. With such an operator, the application can define whether and how the virtual base subobject is assigned.

9.10 Caveats of using protoize

The conversion programs protoize and unprotoize can sometimes change a source file in a way that won't work unless you rearrange it.

- protoize can insert references to a type name or type tag before the definition, or in a file where they are not defined.

  If this happens, compiler error messages should show you where the new references are, so fixing the file by hand is straightforward.
- There are some C constructs which protoize cannot figure out. For example, it can't determine argument types for declaring a pointer-to-function variable; this you must do by hand. protoize inserts a comment containing '???' each time it finds such a variable; so you can find all such variables by searching for this string. ISO C does not require declaring the argument types of pointer-to-function types.
- Using unprotoize can easily introduce bugs. If the program relied on prototypes to bring about conversion of arguments, these conversions will not take place in the program without prototypes. One case in which you can be sure unprotoize is safe is when you are removing prototypes that were made with protoize; if the program worked before without any prototypes, it will work again without them.

  You can find all the places where this problem might occur by compiling the program with the '-Wconversion' option. It prints a warning whenever an argument is converted.
- Both conversion programs can be confused if there are macro calls in and around the text to be converted. In other words, the standard syntax for a declaration or definition

must not result from expanding a macro. This problem is inherent in the design of C and cannot be fixed. If only a few functions have confusing macro calls, you can easily convert them manually.

- `protoize` cannot get the argument types for a function whose definition was not actually compiled due to preprocessing conditionals. When this happens, `protoize` changes nothing in regard to such a function. `protoize` tries to detect such instances and warn about them.

  You can generally work around this problem by using `protoize` step by step, each time specifying a different set of '`-D`' options for compilation, until all of the functions have been converted. There is no automatic way to verify that you have got them all, however.

- Confusion may result if there is an occasion to convert a function declaration or definition in a region of source code where there is more than one formal parameter list present. Thus, attempts to convert code containing multiple (conditionally compiled) versions of a single function header (in the same vicinity) may not produce the desired (or expected) results.

  If you plan on converting source files which contain such code, it is recommended that you first make sure that each conditionally compiled region of source code which contains an alternative function header also contains at least one additional follower token (past the final right parenthesis of the function header). This should circumvent the problem.

- `unprotoize` can become confused when trying to convert a function definition or declaration which contains a declaration for a pointer-to-function formal argument which has the same name as the function being defined or declared. We recommend you avoid such choices of formal parameter names.

- You might also want to correct some of the indentation by hand and break long lines. (The conversion programs don't write lines longer than eighty characters in any case.)

9.11 Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GCC is better without them.

- Checking the number and type of arguments to a function which has an old-fashioned definition and no prototype.

  Such a feature would work only occasionally—only for calls that appear in the same file as the called function, following the definition. The only way to check all calls reliably is to add a prototype for the function. But adding a prototype eliminates the motivation for this feature. So the feature is not worthwhile.

- Warning about using an expression whose type is signed as a shift count.

  Shift count operands are probably signed more often than unsigned. Warning about this would cause far more annoyance than good.

- Warning about assigning a signed value to an unsigned variable.

  Such assignments must be very common; warning about them would cause more annoyance than good.

- Warning when a non-void function value is ignored.

  Coming as I do from a Lisp background, I balk at the idea that there is something dangerous about discarding a value. There are functions that return values which some callers may find useful; it makes no sense to clutter the program with a cast to `void` whenever the value isn't useful.

- Making '`-fshort-enums`' the default.

  This would cause storage layout to be incompatible with most other C compilers. And it doesn't seem very important, given that you can get the same result in other ways. The case where it matters most is when the enumeration-valued object is inside a structure, and in that case you can specify a field width explicitly.

- Making bit-fields unsigned by default on particular machines where "the ABI standard" says to do so.

  The ISO C standard leaves it up to the implementation whether a bit-field declared plain `int` is signed or not. This in effect creates two alternative dialects of C.

  The GNU C compiler supports both dialects; you can specify the signed dialect with '`-fsigned-bitfields`' and the unsigned dialect with '`-funsigned-bitfields`'. However, this leaves open the question of which dialect to use by default.

  Currently, the preferred dialect makes plain bit-fields signed, because this is simplest. Since `int` is the same as `signed int` in every other context, it is cleanest for them to be the same in bit-fields as well.

  Some computer manufacturers have published Application Binary Interface standards which specify that plain bit-fields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bit-fields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bit-fields or unsigned is of no concern to other object files, even if they access the same bit-fields in the same data structures.

  A given program is written in one or the other of these two dialects. The program stands a chance to work on most any machine if it is compiled with the proper dialect. It is unlikely to work at all if compiled with the wrong dialect.

  Many users appreciate the GNU C compiler because it provides an environment that is uniform across machines. These users would be inconvenienced if the compiler treated plain bit-fields differently on certain machines.

  Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU C compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility.

  This is why GCC does and will treat plain bit-fields in the same fashion on all types of machines (by default).

  There are some arguments for making bit-fields unsigned by default on all machines. If, for example, this becomes a universal de facto standard, it would make sense for GCC to go along with it. This is something to be considered in the future.

(Of course, users strongly concerned about portability should indicate explicitly in each bit-field whether it is signed or not. In this way, they write programs which have the same meaning in both C dialects.)

- Undefining `__STDC__` when '`-ansi`' is not used.

  Currently, GCC defines `__STDC__` as long as you don't use '`-traditional`'. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ISO C, such as function prototypes or ISO token concatenation. Since plain `gcc` supports all the features of ISO C, the correct answer to these questions is "yes".

  Some users try to use `__STDC__` to check for the availability of certain library facilities. This is actually incorrect usage in an ISO C program, because the ISO C standard says that a conforming freestanding implementation should define `__STDC__` even though it does not have the library facilities. '`gcc -ansi -pedantic`' is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ISO C library.

  Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ISO C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that claim to support ISO C, such as '`gcc -ansi`'— not for other compilers such as plain `gcc`. Whatever the ISO C standard says is relevant to the design of plain `gcc` without '`-ansi`' only for pragmatic reasons, not as a requirement.

  GCC normally defines `__STDC__` to be 1, and in addition defines `__STRICT_ANSI__` if you specify the '`-ansi`' option, or a '`-std`' option for strict conformance to some version of ISO C. On some hosts, system include files use a different convention, where `__STDC__` is normally 0, but is 1 if the user specifies strict conformance to the C Standard. GCC follows the host convention when processing system include files, but when processing user files it follows the usual GNU C convention.

- Undefining `__STDC__` in C++.

  Programs written to compile with C++-to-C translators get the value of `__STDC__` that goes with the C compiler that is subsequently used. These programs must test `__STDC__` to determine what kind of C preprocessor that compiler uses: whether they should concatenate tokens in the ISO C fashion or in the traditional fashion.

  These programs work properly with GNU C++ if `__STDC__` is defined. They would not work otherwise.

  In addition, many header files are written to provide prototypes in ISO C but not in traditional C. Many of these header files can work without change in C++ provided `__STDC__` is defined. If `__STDC__` is not defined, they will all fail, and will all need to be changed to test explicitly for C++ as well.

- Deleting "empty" loops.

  Historically, GCC has not deleted "empty" loops under the assumption that the most likely reason you would put one in a program is to have a delay, so deleting them will not make real programs run any faster.

  However, the rationale here is that optimization of a nonempty loop cannot produce an empty one, which holds for C but is not always the case for C++.

Moreover, with '-funroll-loops' small "empty" loops are already removed, so the current behavior is both sub-optimal and inconsistent and will change in the future.

- Making side effects happen in the same order as in some other compiler.

  It is never safe to depend on the order of evaluation of side effects. For example, a function call like this may very well behave differently from one compiler to another:

  ```
  void func (int, int);

  int i = 2;
  func (i++, i++);
  ```

  There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. `func` might get the arguments '2, 3', or it might get '3, 2', or even '2, 2'.

- Not allowing structures with volatile fields in registers.

  Strictly speaking, there is no prohibition in the ISO C standard against allowing structures with volatile fields in registers, but it does not seem to make any sense and is probably not what you wanted to do. So the compiler will give an error message in this case.

- Making certain warnings into errors by default.

  Some ISO C testsuites report failure when the compiler does not produce an error message for a certain program.

  ISO C requires a "diagnostic" message for certain kinds of invalid programs, but a warning is defined by GCC to count as a diagnostic. If GCC produces a warning but not an error, that is correct ISO C support. If test suites call this "failure", they should be run with the GCC option '-pedantic-errors', which will turn these warnings into errors.

9.12 Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

> *Errors* report problems that make it impossible to compile your program. GCC reports errors with the source file name and line number where the problem is apparent.
>
> *Warnings* report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text 'warning:' to distinguish them from error messages.

Warnings may indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU C or C++. Many warnings are issued only if you ask for them, with one of the '-W' options (for instance, '-Wall' requests a variety of useful warnings).

GCC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The '-pedantic' option tells GCC to issue warnings in such cases; '`-pedantic-errors`' says to make them errors instead. This does not mean that *all* non-ISO constructs get warnings or errors.

See Section 3.8 [Options to Request or Suppress Warnings], page 31, for more detail on these and related command-line options.

10 Reporting Bugs

Your bug reports play an essential role in making GCC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 9 [Trouble], page 273. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. (If it does not, look in the service directory; see Chapter 11 [Service], page 299.) In any case, the principal function of a bug report is to help the entire community by making the next version of GCC work better. Bug reports are your contribution to the maintenance of GCC.

Since the maintainers are very overloaded, we cannot respond to every bug report. However, if the bug has not been fixed, we are likely to send you a patch and ask you to tell us whether it works.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

10.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C and traditional C (see Section 9.5 [Incompatibilities], page 278). These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C or C++ compiler.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results. But the value of the function is undefined if `return` is omitted; it is not a bug when GCC produces different results.

  Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GCC might interpret it another way. Neither compiler is wrong. The bug is in your code.

  After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.
- If the compiler produces an error message for valid input, that is a compiler bug.

- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of one of the languages GCC supports, your suggestions for improvement of GCC are welcome in any case.

10.2 Where to Report Bugs

Send bug reports for the GNU Compiler Collection to `gcc-bugs@gcc.gnu.org`. In accordance with the GNU-wide convention, in which bug reports for tool "foo" are sent to '`bug-foo@gnu.org`', the address `bug-gcc@gnu.org` may also be used; it will forward to the address given above.

Please read `http://gcc.gnu.org/bugs.html` for additional and/or more up-to-date bug reporting instructions before you post a bug report.

10.3 How to Report Bugs

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

To enable someone to investigate the bug, you should include all these things:

- The version of GCC. You can get this by running it with the '`-v`' option.

  Without this, we won't know whether there is any point in looking for the bug in the current version of GCC.

- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send a source file and any header files that it requires. If the bug is in the compiler proper ('`cc1`'), send the preprocessor output generated by adding '`-save-temps`' to the compilation command (see Section 3.9 [Debugging Options], page 43). When you do this, use the same '`-I`', '`-D`' or '`-U`' options that you used in actual compilation. Then send the *input*.i or *input*.ii files generated.

  A single statement is not enough of an example. In order to compile it, it must be embedded in a complete file of compiler input; and the bug might depend on the details of how this is done.

  Without a real example one can compile, all anyone can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.

  Even if the input file that fails comes from a GNU program, you should still send the complete test case. Don't ask the GCC maintainers to do the extra work of obtaining the program in question—they are all overworked as it is. Also, the problem may depend on what is in the header files on your system; it is unreliable for the GCC maintainers to try the problem with the header files available to them. By sending CPP output, you can eliminate this source of uncertainty and save us a certain percentage of wild goose chases.
- The command arguments you gave GCC to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list all the options.

  If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.
- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed the compiler.
- A complete list of any modifications you have made to the compiler source. (We don't promise to investigate the bug unless it happens in an unmodified compiler. But if you've made modifications and don't tell us, then you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own (such as a machine description for a machine we don't support) is a modification of the compiler source.
- Details of any other deviations from the standard procedure for installing GCC.
- A description of what behavior you observe that you believe is incorrect. For example, "The compiler gets a fatal signal," or, "The assembler instruction at line 208 in the output is incorrect."

  Of course, if the bug is that the compiler gets a fatal signal, then one can't miss it. But if the bug is incorrect output, the maintainer might not notice unless it is glaringly wrong. None of us has time to study all the assembler code from a 50-line C program just on the chance that one instruction might be wrong. We need *you* to do this part!

Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here would not. If you *said* to expect a crash, then when the compiler here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

If the problem is a diagnostic when compiling GCC with some other compiler, say whether it is a warning or an error.

Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information unless the program is short and simple. None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell us which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as finding a bug in the program itself.

- If you send examples of assembler code output from GCC, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to mention something in the GCC source, refer to it by context, not by line number.

  The line numbers in the development sources don't match those in your sources. Your line numbers would convey no useful information to the maintainers.
- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to have any chance of being useful.

  For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GCC because the compiler is largely data-driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

  Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

  In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop—which insn it has reached—is usually in a local variable, not in an argument.

  What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GCC bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions may affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.
- In particular, some people insert conditionals '`#ifdef BUG`' around a statement which, if removed, makes the bug not happen. These are just clutter; we won't pay any attention to them anyway. Besides, you should send us cpp output, and that can't have conditionals.
- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GCC it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send the example, we won't be able to construct one, so we won't be able to verify that the bug is fixed.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we won't install it. A test case will help us to understand.

  See `http://gcc.gnu.org/contribute.html` for guidelines on how to make it easy for us to understand and install your patches.
- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without first using the debugger to find the facts.
- A core dump file.

  We have no way of examining a core dump for your type of machine unless we have an identical system—and if we do have one, we should be able to reproduce the crash ourselves.

10.4 The gccbug script

To simplify creation of bug reports, and to allow better tracking of reports, we use the GNATS bug tracking system. Part of that system is the `gccbug` script. This is a Unix shell script, so you need a shell to run it. It is normally installed in the same directory where `gcc` is installed.

The gccbug script is derived from send-pr, see section “Creating new Problem Reports” in *Reporting Problems*. When invoked, it starts a text editor so you can fill out the various fields of the report. When the you quit the editor, the report is automatically send to the bug reporting address.

A number of fields in this bug report form are specific to GCC, and are explained at `http://gcc.gnu.org/gnats.html`.

11 How To Get Help with GCC

If you need help installing, using or changing GCC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try gcc-help@gcc.gnu.org (for help installing or using GCC), and if that brings no response, try gcc@gcc.gnu.org. For help changing GCC, ask gcc@gcc.gnu.org. If you think you have found a bug in GCC, please report it following the instructions at see Section 10.3 [Bug Reporting], page 294.
- Look in the service directory for someone who might help you for a fee. The service directory is found at http://www.gnu.org/prep/service.html.

12 Contributing to GCC Development

If you would like to help pretest GCC releases to assure they work well, our current development sources are available by CVS (see `http://gcc.gnu.org/cvs.html`). Source and binary snapshots are also available for FTP; see `http://gcc.gnu.org/snapshots.html`.

If you would like to work on improvements to GCC, please read the advice at these URLs:

```
http://gcc.gnu.org/contribute.html
http://gcc.gnu.org/contributewhy.html
```

for information on how to make useful contributions and avoid duplication of effort. Suggested projects are listed at `http://gcc.gnu.org/projects/`.

13 Using GCC on VMS

Here is how to use GCC on VMS.

13.1 Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GCC attempts to translate file names in '#include' into names that VMS will understand. The basic strategy is to prepend a prefix to the specification of the include file, convert the whole filename to a VMS filename, and then try to open the file. GCC tries various prefixes one by one until one of them succeeds:

1. The first prefix is the 'GNU_CC_INCLUDE:' logical name: this is where GNU C header files are traditionally stored. If you wish to store header files in non-standard locations, then you can assign the logical 'GNU_CC_INCLUDE' to be a search list, where each element of the list is suitable for use with a rooted logical.
2. The next prefix tried is 'SYS$SYSROOT:[SYSLIB.]'. This is where VAX-C header files are traditionally stored.
3. If the include file specification by itself is a valid VMS filename, the preprocessor then uses this name with no prefix in an attempt to open the include file.
4. If the file specification is not a valid VMS filename (i.e. does not contain a device or a directory specifier, and contains a '/' character), the preprocessor tries to convert it from Unix syntax to VMS syntax.

   Conversion works like this: the first directory name becomes a device, and the rest of the directories are converted into VMS-format directory names. For example, the name 'X11/foobar.h' is translated to 'X11:[000000]foobar.h' or 'X11:foobar.h', whichever one can be opened. This strategy allows you to assign a logical name to point to the actual location of the header files.
5. If none of these strategies succeeds, the '#include' fails.

Include directives of the form:

```
#include foobar
```

are a common source of incompatibility between VAX-C and GCC. VAX-C treats this much like a standard #include <foobar.h> directive. That is incompatible with the ISO C behavior implemented by GCC: to expand the name foobar as a macro. Macro expansion should eventually yield one of the two standard formats for #include:

```
#include "file"
#include <file>
```

If you have this problem, the best solution is to modify the source to convert the #include directives to one of the two standard forms. That will work with either compiler. If you want a quick and dirty fix, define the file names as macros with the proper expansion, like this:

```
#define stdio <stdio.h>
```

This will work, as long as the name doesn't conflict with anything else in the program.

Another source of incompatibility is that VAX-C assumes that:

```
#include "foobar"
```

is actually asking for the file 'foobar.h'. GCC does not make this assumption, and instead takes what you ask for literally; it tries to read the file 'foobar'. The best way to avoid this problem is to always specify the desired file extension in your include directives.

GCC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GCC distribution does not contain header files to define constants and structures for some VMS system-specific functions, there is no reason why you cannot use GCC with any of these functions. You first may have to generate or create header files, either by using the public domain utility UNSDL (which can be found on a DECUS tape), or by extracting the relevant modules from one of the system macro libraries, and using an editor to construct a C header file.

A #include file name cannot contain a DECNET node name. The preprocessor reports an I/O error if you attempt to use a node name, whether explicitly, or implicitly via a logical name.

13.2 Global Declarations and VMS

GCC does not provide the globalref, globaldef and globalvalue keywords of VAX-C. You can get the same effect with an obscure feature of GAS, the GNU assembler. (This requires GAS version 1.39 or later.) The following macros allow you to use this feature in a fairly natural way:

```
#ifdef __GNUC__
#define GLOBALREF(TYPE,NAME)                       \
  TYPE NAME                                        \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME)
#define GLOBALDEF(TYPE,NAME,VALUE)                 \
  TYPE NAME                                        \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME) \
    = VALUE
#define GLOBALVALUEREF(TYPE,NAME)                  \
  const TYPE NAME[1]                               \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)
#define GLOBALVALUEDEF(TYPE,NAME,VALUE)            \
  const TYPE NAME[1]                               \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)  \
    = {VALUE}
#else
#define GLOBALREF(TYPE,NAME) \
  globalref TYPE NAME
#define GLOBALDEF(TYPE,NAME,VALUE) \
  globaldef TYPE NAME = VALUE
#define GLOBALVALUEDEF(TYPE,NAME,VALUE) \
  globalvalue TYPE NAME = VALUE
#define GLOBALVALUEREF(TYPE,NAME) \
  globalvalue TYPE NAME
#endif
```

(The `_$$PsectAttributes_GLOBALSYMBOL` prefix at the start of the name is removed by the assembler, after it has modified the attributes of the symbol). These macros are provided in the VMS binaries distribution in a header file 'GNU_HACKS.H'. An example of the usage is:

```
GLOBALREF (int, ijk);
GLOBALDEF (int, jkl, 0);
```

The macros `GLOBALREF` and `GLOBALDEF` cannot be used straightforwardly for arrays, since there is no way to insert the array dimension into the declaration at the right place. However, you can declare an array with these macros if you first define a typedef for the array type, like this:

```
typedef int intvector[10];
GLOBALREF (intvector, foo);
```

Array and structure initializers will also break the macros; you can define the initializer to be a macro of its own, or you can expand the `GLOBALDEF` macro by hand. You may find a case where you wish to use the `GLOBALDEF` macro with a large array, but you are not interested in explicitly initializing each element of the array. In such cases you can use an initializer like: `{0,}`, which will initialize the entire array to `0`.

A shortcoming of this implementation is that a variable declared with `GLOBALVALUEREF` or `GLOBALVALUEDEF` is always an array. For example, the declaration:

```
GLOBALVALUEREF(int, ijk);
```

declares the variable `ijk` as an array of type `int [1]`. This is done because a globalvalue is actually a constant; its "value" is what the linker would normally consider an address. That is not how an integer value works in C, but it is how an array works. So treating the symbol as an array name gives consistent results—with the exception that the value seems to have the wrong type. **Don't try to access an element of the array.** It doesn't have any elements. The array "address" may not be the address of actual storage.

The fact that the symbol is an array may lead to warnings where the variable is used. Insert type casts to avoid the warnings. Here is an example; it takes advantage of the ISO C feature allowing macros that expand to use the same name as the macro itself.

```
GLOBALVALUEREF (int, ss$_normal);
GLOBALVALUEDEF (int, xyzzy,123);
#ifdef __GNUC__
#define ss$_normal ((int) ss$_normal)
#define xyzzy ((int) xyzzy)
#endif
```

Don't use `globaldef` or `globalref` with a variable whose type is an enumeration type; this is not implemented. Instead, make the variable an integer, and use a `globalvaluedef` for each of the enumeration values. An example of this would be:

```
#ifdef __GNUC__
GLOBALDEF (int, color, 0);
GLOBALVALUEDEF (int, RED, 0);
GLOBALVALUEDEF (int, BLUE, 1);
GLOBALVALUEDEF (int, GREEN, 3);
#else
enum globaldef color {RED, BLUE, GREEN = 3};
#endif
```

13.3 Other VMS Issues

GCC automatically arranges for `main` to return 1 by default if you fail to specify an explicit return value. This will be interpreted by VMS as a status code indicating a normal successful completion. Version 1 of GCC did not provide this default.

GCC on VMS works only with the GNU assembler, GAS. You need version 1.37 or later of GAS in order to produce value debugging information for the VMS debugger. Use the ordinary VMS linker with the object files produced by GAS.

Under previous versions of GCC, the generated code would occasionally give strange results when linked to the sharable '`VAXCRTL`' library. Now this should work.

A caveat for use of `const` global variables: the `const` modifier must be specified in every external declaration of the variable in all of the source files that use that variable. Otherwise the linker will issue warnings about conflicting attributes for the variable. Your program will still work despite the warnings, but the variable will be placed in writable storage.

Although the VMS linker does distinguish between upper and lower case letters in global symbols, most VMS compilers convert all such symbols into upper case and most run-time library routines also have upper case names. To be able to reliably call such routines, GCC (by means of the assembler GAS) converts global symbols into upper case like other VMS compilers. However, since the usual practice in C is to distinguish case, GCC (via GAS) tries to preserve usual C behavior by augmenting each name that is not all lower case. This means truncating the name to at most 23 characters and then adding more characters at the end which encode the case pattern of those 23. Names which contain at least one dollar sign are an exception; they are converted directly into upper case without augmentation.

Name augmentation yields bad results for programs that use precompiled libraries (such as Xlib) which were generated by another compiler. You can use the compiler option '`/NOCASE_HACK`' to inhibit augmentation; it makes external C functions and variables case-independent as is usual on VMS. Alternatively, you could write all references to the functions and variables in such libraries using lower case; this will work on VMS, but is not portable to other systems. The compiler option '`/NAMES`' also provides control over global name handling.

Function and variable names are handled somewhat differently with G++. The GNU C++ compiler performs *name mangling* on function names, which means that it adds information to the function name to describe the data types of the arguments that the function takes. One result of this is that the name of a function can become very long. Since the VMS linker only recognizes the first 31 characters in a name, special action is taken to ensure that each function and variable has a unique name that can be represented in 31 characters.

If the name (plus a name augmentation, if required) is less than 32 characters in length, then no special action is performed. If the name is longer than 31 characters, the assembler (GAS) will generate a hash string based upon the function name, truncate the function name to 23 characters, and append the hash string to the truncated name. If the '`/VERBOSE`' compiler option is used, the assembler will print both the full and truncated names of each symbol that is truncated.

The '`/NOCASE_HACK`' compiler option should not be used when you are compiling programs that use libg++. libg++ has several instances of objects (i.e. `Filebuf` and `filebuf`) which become indistinguishable in a case-insensitive environment. This leads to cases where you need to inhibit augmentation selectively (if you were using libg++ and Xlib in the same program, for example). There is no special feature for doing this, but you can get the result by defining a macro for each mixed case symbol for which you wish to inhibit augmentation. The macro should expand into the lower case equivalent of itself. For example:

```
#define StuDlyCapS studlycaps
```

These macro definitions can be placed in a header file to minimize the number of changes to your source code.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

The GNU Project and GNU/Linux

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced "guh-NEW".) Variants of the GNU operating system, which use the kernel Linux, are now widely used; though these systems are often referred to as "Linux", they are more accurately called GNU/Linux systems.

For more information, see:

```
http://www.gnu.org/
http://www.gnu.org/gnu/linux-and-gnu.html
```

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. 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.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

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

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. 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.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant

Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Contributors to GCC

The GCC project would like to thank its many contributors. Without them the project would not have been nearly as successful as it has been. Any omissions in this list are accidental. Feel free to contact `law@redhat.com` if you have been left out or some of your contributions are not listed. Please keep this list in alphabetical order.

Some projects operating under the GCC project maintain their own list of contributors, such as the C++ library (`http://gcc.gnu.org/libstdc++/`).

- Analog Devices helped implement the support for complex data types and iterators.
- John David Anglin for improvements to libstdc++-v3 and the HP-UX port.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Alasdair Baird for various bugfixes.
- Gerald Baumgartner added the signature extension to the C++ front end.
- Neil Booth for work on cpplib, lang hooks, debug hooks and other miscellaneous clean-ups.
- Per Bothner for his direction via the steering committee and various improvements to our infrastructure for supporting new languages. Chill and Java front end implementations. Initial implementations of cpplib, fix-header, config.guess, libio, and past C++ library (libg++) maintainer.
- Devon Bowen helped port GCC to the Tahoe.
- Don Bowman for mips-vxworks contributions.
- Dave Brolley for work on cpplib and Chill.
- Robert Brown implemented the support for Encore 32000 systems.
- Christian Bruel for improvements to local store elimination.
- Herman A.J. ten Brugge for various fixes.
- Joe Buck for his direction via the steering committee.
- Craig Burley for leadership of the Fortran effort.
- Paolo Carlini for his work on libstdc++-v3.
- John Carr for his alias work, SPARC hacking, infrastructure improvements, previous contributions to the steering committee, loop optimizations, etc.
- Steve Chamberlain wrote the support for the Hitachi SH and H8 processors and the PicoJava processor.
- Scott Christley for his Objective-C contributions.
- Branko Cibej for more warning contributions.
- Nick Clifton for arm, mcore, fr30, v850, m32r work, '`--help`', and other random hacking.
- Ralf Corsepius for SH testing and minor bugfixing.
- Stan Cox for care and feeding of the x86 port and lots of behind the scenes hacking.
- Alex Crain provided changes for the 3b1.
- Ian Dall for major improvements to the NS32k port.

- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Ulrich Drepper for his work on the C++ runtime libraries, glibc, testing of GCC using glibc, ISO C99 support, CFG dumping support, etc.
- Richard Earnshaw for his ongoing work with the ARM.
- David Edelsohn for his direction via the steering committee, ongoing work with the RS6000/PowerPC port, and help cleaning up Haifa loop changes.
- Paul Eggert for random hacking all over GCC.
- Mark Elbrecht for various DJGPP improvements.
- Ben Elliston for his work to move the Objective-C runtime into its own subdirectory and for his work on autoconf.
- Marc Espie for OpenBSD support.
- Doug Evans for much of the global optimization framework, arc, m32r, and SPARC work.
- Fred Fish for BeOS support and Ada fixes.
- Peter Gerwinski for various bugfixes and the Pascal front end.
- Kaveh Ghazi for his direction via the steering committee and amazing work to make '`-W -Wall`' useful.
- Judy Goldberg for c++ contributions.
- Torbjorn Granlund for various fixes and the c-torture testsuite, multiply- and divide-by-constant optimization, improved long long support, improved leaf function register allocation, and his direction via the steering committee.
- Anthony Green for his '`-Os`' contributions and Java front end work.
- Michael K. Gschwind contributed the port to the PDP-11.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Bruno Haible for improvements in the runtime overhead for EH, new warnings and assorted bugfixes.
- Andrew Haley for his Java work.
- Chris Hanson assisted in making GCC work on HP-UX for the 9000 series 300.
- Michael Hayes for various thankless work he's done trying to get the c30/c40 ports functional. Lots of loop and unroll improvements and fixes.
- Kate Hedstrom for staking the g77 folks with an initial testsuite.
- Richard Henderson for his ongoing SPARC, alpha, and ia32 work, loop opts, and generally fixing lots of old problems we've ignored for years, flow rewrite and lots of further stuff, including reviewing tons of patches.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Manfred Hollstein for his ongoing work to keep the m88k alive, lots of testing an bugfixing, particularly of our configury code.

- Steve Holmgren for MachTen patches.
- Jan Hubicka for his x86 port improvements.
- Christian Iseli for various bugfixes.
- Kamil Iskra for general m68k hacking.
- Lee Iverson for random fixes and MIPS testing.
- Andreas Jaeger for various fixes to the MIPS port
- Jakub Jelinek for his SPARC work and sibling call optimizations as well as lots of bug fixes and test cases.
- Janis Johnson for ia64 testing and fixes and for her quality improvement sidetracks.
- J. Kean Johnston for OpenServer support.
- Klaus Kaempf for his ongoing work to make alpha-vms a viable target.
- David Kashtan of SRI adapted GCC to VMS.
- Geoffrey Keating for his ongoing work to make the PPC work for GNU/Linux and his automatic regression tester.
- Brendan Kehoe for his ongoing work with g++.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination and delay slot scheduling. Richard Kenner was also the head maintainer of GCC for several years.
- Mumit Khan for various contributions to the cygwin and mingw32 ports and maintaining binary releases for Windows hosts.
- Robin Kirkham for cpu32 support.
- Mark Klein for PA improvements.
- Thomas Koenig for various bugfixes.
- Bruce Korb for the new and improved fixincludes code.
- Benjamin Kosnik for his g++ work and for leading the libstdc++-v3 effort.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Jeff Law for his direction via the steering committee, coordinating the entire egcs project and GCC 2.95, rolling out snapshots and releases, handling merges from GCC2, reviewing tons of patches that might have fallen through the cracks else, and random but extensive hacking.
- Marc Lehmann for his direction via the steering committee and helping with analysis and improvements of x86 performance.
- Ted Lemon wrote parts of the RTL reader and printer.
- Kriang Lerdsuwanakij for improvements to demangler and various c++ fixes.
- Warren Levy major work on libgcj (Java Runtime Library) and random work on the Java front end.

- Alain Lichnewsky ported GCC to the MIPS CPU.
- Robert Lipe for OpenServer support, new testsuites, testing, etc.
- Weiwen Liu for testing and various bugfixes.
- Dave Love for his ongoing work with the Fortran front end and runtime libraries.
- Martin von Löwis for internal consistency checking infrastructure, and various C++ improvements including namespace support.
- H.J. Lu for his previous contributions to the steering committee, many x86 bug reports, prototype patches, and keeping the GNU/Linux ports working.
- Greg McGary for random fixes and (someday) bounded pointers.
- Andrew MacLeod for his ongoing work in building a real EH system, various code generation improvements, work on the global optimizer, etc.
- Vladimir Makarov for hacking some ugly i960 problems, PowerPC hacking improvements to compile-time performance and overall knowledge and direction in the area of instruction scheduling.
- Bob Manson for his behind the scenes work on dejagnu.
- Michael Meissner for LRS framework, ia32, m32r, v850, m88k, MIPS, powerpc, haifa, ECOFF debug support, and other assorted hacking.
- Jason Merrill for his direction via the steering committee and leading the g++ effort.
- David Miller for his direction via the steering committee, lots of SPARC work, improvements in jump.c and interfacing with the Linux kernel developers.
- Gary Miller ported GCC to Charles River Data Systems machines.
- Mark Mitchell for his direction via the steering committee, mountains of C++ work, load/store hoisting out of loops, alias analysis improvements, ISO C `restrict` support, and serving as release manager for GCC 3.x.
- Alan Modra for various GNU/Linux bits and testing.
- Toon Moene for his direction via the steering committee, Fortran maintenance, and his ongoing work to make us make Fortran run fast.
- Jason Molenda for major help in the care and feeding of all the services on the gcc.gnu.org (formerly egcs.cygnus.com) machine—mail, web services, ftp services, etc etc.
- Catherine Moore for fixing various ugly problems we have sent her way, including the haifa bug which was killing the Alpha & PowerPC Linux kernels.
- David Mosberger-Tang for various Alpha improvements.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits and for ISO C99 support.
- Bill Moyer for his behind the scenes work on various issues.
- Philippe De Muyter for his work on the m68k port.
- Joseph S. Myers for his work on the PDP-11 port, format checking and ISO C99 support, and continuous emphasis on (and contributions to) documentation.
- Nathan Myers for his work on libstdc++-v3.

- NeXT, Inc. donated the front end that supports the Objective-C language.
- Hans-Peter Nilsson for the CRIS and MMIX ports, improvements to the search engine setup, various documentation fixes and other small fixes.
- Geoff Noer for this work on getting cygwin native builds working.
- David O'Brien for the FreeBSD/alpha, FreeBSD/AMD x86-64, FreeBSD/ARM, FreeBSD/PowerPC, and FreeBSD/SPARC64 ports and related infrastructure improvements.
- Alexandre Oliva for various build infrastructure improvements, scripts and amazing testing work.
- Melissa O'Neill for various NeXT fixes.
- Rainer Orth for random MIPS work, including improvements to our o32 ABI support, improvements to dejagnu's MIPS support, etc.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Alexandre Petit-Bianco for his Java work.
- Matthias Pfaller for major improvements to the NS32k port.
- Gerald Pfeifer for his direction via the steering committee, pointing out lots of problems we need to solve, maintenance of the web pages, and taking care of documentation maintenance in general.
- Ovidiu Predescu for his work on the Objective-C front end and runtime libraries.
- Ken Raeburn for various improvements to checker, MIPS ports and various cleanups in the compiler.
- David Reese of Sun Microsystems contributed to the Solaris on PowerPC port.
- Gabriel Dos Reis for contributions and maintenance of libstdc++-v3, including valarray implementation and limits support.
- Joern Rennecke for maintaining the sh port, loop, regmove & reload hacking.
- Loren J. Rittle for improvements to libstdc++-v3 and the FreeBSD port.
- Craig Rodrigues for processing tons of bug reports.
- Gavin Romig-Koch for lots of behind the scenes MIPS work.
- Ken Rose for fixes to our delay slot filling code.
- Paul Rubin wrote most of the preprocessor.
- Juha Sarlin for improvements to the H8 code generator.
- Greg Satz assisted in making GCC work on HP-UX for the 9000 series 300.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- William Schelter did most of the work on the Intel 80386 support.
- Bernd Schmidt for various code generation improvements and major work in the reload pass as well a serving as release manager for GCC 2.95.3.
- Andreas Schwab for his work on the m68k port.
- Joel Sherrill for his direction via the steering committee, RTEMS contributions and RTEMS testing.
- Nathan Sidwell for many C++ fixes/improvements.

- Jeffrey Siegal for helping RMS with the original design of GCC, some code which handles the parse tree and RTL data structures, constant folding and help with the original VAX & m68k ports.
- Franz Sirl for his ongoing work with making the PPC port stable for linux.
- Andrey Slepuhin for assorted AIX hacking.
- Christopher Smith did the port for Convex machines.
- Randy Smith finished the Sun FPA support.
- Scott Snyder for various fixes.
- Richard Stallman, for writing the original gcc and launching the GNU project.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Nigel Stephens for various mips16 related fixes/improvements.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Graham Stott for various infrastructure improvements.
- Mike Stump for his Elxsi port, g++ contributions over the years and more recently his vxworks contributions
- Shigeya Suzuki for this fixes for the bsdi platforms.
- Ian Lance Taylor for his mips16 work, general configury hacking, fixincludes, etc.
- Holger Teutsch provided the support for the Clipper CPU.
- Gary Thomas for his ongoing work to make the PPC work for GNU/Linux.
- Philipp Thomas for random bugfixes throughout the compiler
- Kresten Krab Thorup wrote the run time support for the Objective-C language.
- Michael Tiemann for random bugfixes, the first instruction scheduler, initial C++ support, function integration, NS32k, SPARC and M88k machine description work, delay slot scheduling.
- Teemu Torma for thread safe exception handling support.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the VAX machine description.
- Tom Tromey for internationalization support and his Java work.
- Lassi Tuura for improvements to config.guess to determine HP processor types.
- Todd Vierling for contributions for NetBSD ports.
- Dean Wakerley for converting the install documentation from HTML to texinfo in time for GCC 3.0.
- Krister Walfridsson for random bugfixes.
- John Wehle for various improvements for the x86 code generator, related infrastructure improvements to help x86 code generation, value range propagation and other work, WE32k port.
- Zack Weinberg for major work on cpplib and various other bugfixes.
- Dale Wiles helped port GCC to the Tahoe.
- Bob Wilson from Tensilica, Inc. for the Xtensa port.

- Jim Wilson for his direction via the steering committee, tackling hard problems in various places that nobody else wanted to work on, strength reduction and other loop optimizations.
- Carlo Wood for various fixes.
- Tom Wood for work on the m88k port.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- Kevin Zachmann helped ported GCC to the Tahoe.

We'd also like to thank the folks who have contributed time and energy in testing GCC:

- David Billinghurst
- Horst von Brand
- Rodney Brown
- Joe Buck
- Craig Burley
- Ulrich Drepper
- David Edelsohn
- Yung Shing Gene
- Kaveh Ghazi
- Kate Hedstrom
- Richard Henderson
- Manfred Hollstein
- Kamil Iskra
- Christian Joensson
- Jeff Law
- Robert Lipe
- Damon Love
- Dave Love
- H.J. Lu
- Brad Lucier
- Mumit Khan
- Matthias Klose
- Martin Knoblauch
- David Miller
- Toon Moene
- Matthias Mueller
- Alexandre Oliva
- Richard Polton
- David Rees
- Loren J. Rittle

- Peter Schmid
- David Schuler
- Vin Shelton
- Franz Sirl
- Mike Stump
- Carlo Wood
- And many others

And finally we'd like to thank everyone who uses the compiler, submits bug reports and generally reminds us why we're doing this work in the first place.

Option Index

GCC's command line options are indexed here without any initial '-' or '--'. Where an option has both positive and negative forms (such as '-f*option*' and '-fno-*option*'), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Index

0

A

B

C

G

H

I

J

K

L

M

N

O

T

U

V

W

X

Z