Red Hat Enterprise Linux 4

Red Hat Enterprise Linux 4

Red Hat, Inc.

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

gcc(EN)-4-Print-RHI (2005-02-01)

Table of Contents

Red Hat Enterprise Linux 4

Chapter 1. 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 3.4.4. 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.

Chapter 2.

Programming Languages Supported by GCC

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for several major programming languages. These languages currently include C, C++, Objective-C, Java, Fortran, and Ada.

The abbreviation *GCC* has multiple meanings in common use. The current official meaning is "GNU Compiler Collection", which refers generically to the complete suite of tools. The name historically stood for "GNU C Compiler", and this usage is still common when the emphasis is on compiling C programs. Finally, the name is also used when speaking of the *language-independent* component of GCC: code shared among the compilers for all supported languages.

The language-independent component of GCC includes the majority of the optimizers, as well as the "back ends" that generate machine code for various processors.

The part of a compiler that is specific to a particular language is called the "front end". In addition to the front ends that are integrated components of GCC, there are several other front ends that are maintained separately. These support languages such as Pascal, Mercury, and COBOL. To use these, they must be built together with GCC proper.

Most of the compilers for languages other than C have their own names. The C++ compiler is G++, the Ada compiler is GNAT, and so on. When we talk about compiling one of those languages, we might refer to that compiler by its own name, or as GCC. Either is correct.

Historically, compilers for many languages, including C++ and Fortran, have been implemented as "preprocessors" which emit another high level language such as C. None of the compilers included in GCC are implemented this way; they all generate machine code directly. This sort of preprocessor should not be confused with the *C preprocessor*, which is an integral feature of the C, C++, and Objective-C languages.

Chapter 3. 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). Section 4.4 *Options Controlling C Dialect*.

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-3.4/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.

By default, GCC provides some extensions to the C language that on rare occasions conflict with the C standard. Chapter 6 *Extensions to the C Language Family*. 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. Section 4.4 *Options Controlling C Dialect*.

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). Section 11.7 *Standard Libraries*.

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/ is a recent version
- http://www.toodarkpark.org/computers/objc/ is an older example
- http://www.gnustep.org has additional useful information

There is no standard for treelang, which is a sample language front end for GCC. Its only purpose is as a sample for people wishing to write a new language for GCC. The language is documented in `gcc/treelang/treelang.texi` which can be turned into info or HTML format.

Chapter 4.
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.

Section 4.3 *Compiling C++ Programs*, for a summary of special options for compiling C++ programs.

The `gcc` program accepts options and file names as operands. Many options have 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.

4.1. Option Summary

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

Overall Options

Section 4.2 *Options Controlling the Kind of Output*.

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

C Language Options

Section 4.4 *Options Controlling C Dialect*.

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

C++ Language Options

Section 4.5 *Options Controlling C++ Dialect*.

```
-fabi-version=n  -fno-access-control  -fcheck-new
-fconserve-space  -fno-const-strings
-fno-elide-constructors
-fno-enforce-eh-specs
```

```
-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
-fno-threadsafe-statics  -fuse-cxa-atexit  -fno-weak  -nostdinc++
-fno-default-inline  -fvisibility-inlines-hidden
-Wabi  -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

Section 4.6 *Options Controlling Objective-C Dialect*.

```
-fconstant-string-class=class-name
-fgnu-runtime  -fnext-runtime
-fno-nil-receivers
-fobjc-exceptions
-freplace-objc-classes
-fzero-link
-gen-decls
-Wno-protocol  -Wselector -Wundeclared-selector
```

Language Independent Options

Section 4.7 *Options to Control Diagnostic Messages Formatting*.

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

Warning Options

Section 4.8 *Options to Request or Suppress Warnings*.

```
-fsyntax-only  -pedantic  -pedantic-errors
-w  -Wextra  -Wall  -Waggregate-return
-Wcast-align  -Wcast-qual  -Wchar-subscripts  -Wcomment
-Wconversion  -Wno-deprecated-declarations
-Wdisabled-optimization  -Wno-div-by-zero  -Wendif-labels
-Werror  -Werror-implicit-function-declaration
-Wfloat-equal  -Wformat  -Wformat=2
-Wno-format-extra-args -Wformat-nonliteral
-Wformat-security  -Wformat-y2k
-Wimplicit  -Wimplicit-function-declaration  -Wimplicit-int
-Wimport  -Wno-import  -Winit-self  -Winline
-Wno-invalid-offsetof  -Winvalid-pch
-Wlarger-than-len  -Wlong-long
-Wmain  -Wmissing-braces
-Wmissing-format-attribute  -Wmissing-noreturn
-Wno-multichar  -Wnonnull  -Wpacked  -Wpadded
-Wparentheses  -Wpointer-arith  -Wredundant-decls
-Wreturn-type  -Wsequence-point  -Wshadow
-Wsign-compare  -Wstrict-aliasing
-Wswitch  -Wswitch-default  -Wswitch-enum
-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-declarations
-Wmissing-prototypes  -Wnested-externs  -Wold-style-definition
-Wstrict-prototypes  -Wtraditional
-Wdeclaration-after-statement
```

Debugging Options

Section 4.9 *Options for Debugging Your Program or GCC*.

```
-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]
-feliminate-dwarf2-dups -feliminate-unused-debug-types
-feliminate-unused-debug-symbols -fmem-report -fprofile-arcs
-frandom-seed=string -fsched-verbose=n
-ftest-coverage  -ftime-report
-g  -glevel  -gcoff -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

Section 4.10 *Options That Control Optimization*.

```
-falign-functions=n  -falign-jumps=n
-falign-labels=n  -falign-loops=n
-fbranch-probabilities -fprofile-values -fvpt -fbranch-target-load-optimize
-fbranch-target-load-optimize2 -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  -fgcse-las  -floop-optimize
-fcrossjumping  -fif-conversion  -fif-conversion2
-finline-functions  -finline-limit=n  -fkeep-inline-functions
-fkeep-static-consts  -fmerge-constants  -fmerge-all-constants
-fmove-all-movables  -fnew-ra  -fno-branch-count-reg
-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  -ffinite-math-only
-fno-trapping-math  -fno-zero-initialized-in-bss
-fomit-frame-pointer  -foptimize-register-move
-foptimize-sibling-calls  -fprefetch-loop-arrays
-fprofile-generate -fprofile-use
-freduce-all-givs  -fregmove  -frename-registers
-freorder-blocks  -freorder-functions
-frerun-cse-after-loop  -frerun-loop-opt
-frounding-math -fschedule-insns  -fschedule-insns2
-fno-sched-interblock  -fno-sched-spec  -fsched-spec-load
-fsched-spec-load-dangerous
-fsched-stalled-insns=n -sched-stalled-insns-dep=n
-fsched2-use-superblocks
-fsched2-use-traces  -fsignaling-nans
-fsingle-precision-constant
-fstrength-reduce  -fstrict-aliasing  -ftracer  -fthread-jumps
```

```
-funroll-all-loops  -funroll-loops  -fpeel-loops
-funswitch-loops  -fold-unroll-loops  -fold-unroll-all-loops
--param name=value
-O  -O0  -O1  -O2  -O3  -Os
```

Preprocessor Options

Section 4.11 *Options Controlling the Preprocessor*.

```
-Aquestion=answer
-A-question[=answer]
-C  -dD  -dI  -dM  -dN
-Dmacro[=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  -fworking-directory  -remap
-trigraphs  -undef  -Umacro  -Wp,option
-Xpreprocessor option
```

Assembler Option

Section 4.12 *Passing Options to the Assembler*.

```
-Wa,option  -Xassembler option
```

Linker Options

Section 4.13 *Options for Linking*.

```
object-file-name  -llibrary
-nostartfiles  -nodefaultlibs  -nostdlib -pie
-s  -static  -static-libgcc  -shared  -shared-libgcc  -symbolic
-Wl,option  -Xlinker option
-u symbol
```

Directory Options

Section 4.14 *Options for Directory Search*.

```
-Bprefix  -Idir  -I-  -Ldir  -specs=file
```

Target Options

Section 4.16 *Specifying Target Machine and Compiler Version*.

```
-V version  -b machine
```

Machine Dependent Options

Section 4.17 *Hardware Models and Configurations*.

M680x0 Options

```
-m68000  -m68020  -m68020-40  -m68020-60  -m68030  -m68040
-m68060  -mcpu32  -m5200  -m68881  -mbitfield  -mc68000  -mc68020
-mnobitfield  -mrtd  -mshort  -msoft-float  -mpcrel
-malign-int  -mstrict-align  -msep-data  -mno-sep-data
-mshared-library-id=n  -mid-shared-library  -mno-id-shared-library
```

M68hc1x Options

```
-m6811  -m6812  -m68hc11  -m68hc12   -m68hcs12
-mauto-incdec  -minmax  -mlong-calls  -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  -mno-app-regs
-mfaster-structs  -mno-faster-structs
-mflat  -mno-flat  -mfpu  -mno-fpu
-mhard-float  -msoft-float
-mhard-quad-float  -msoft-quad-float
-mimpure-text  -mno-impure-text  -mlittle-endian
-mstack-bias  -mno-stack-bias
-munaligned-doubles  -mno-unaligned-doubles
-mv8plus  -mno-v8plus  -mvis  -mno-vis
-mcypress  -mf930  -mf934
-msparclite  -msupersparc  -mv8
-threads -pthreads
```

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
-mabort-on-noreturn
-mlong-calls  -mno-long-calls
-msingle-pic-base  -mno-single-pic-base
-mpic-register=reg
-mnop-fun-dllimport
-mcirrus-fix-invalid-insns -mno-cirrus-fix-invalid-insns
-mpoke-function-name
-mthumb  -marm
-mtpcs-frame  -mtpcs-leaf-frame
-mcaller-super-interworking  -mcallee-super-interworking
```

MN10300 Options
```
-mmult-bug  -mno-mult-bug
-mam33  -mno-am33
-mam33-2  -mno-am33-2
-mno-crt0  -mrelax
```

M32R/D Options
```
-m32r2 -m32rx -m32r
-mdebug
-malign-loops -mno-align-loops
-missue-rate=number
-mbranch-cost=number
-mmodel=code-size-model-type
-msdata=sdata-type
-mno-flush-func -mflush-func=name
-mno-flush-trap -mflush-trap=number
-G num
```

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
-malign-power  -malign-natural
-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
-mdynamic-no-pic
-mprioritize-restricted-insns=priority
-msched-costly-dep=dependence_type
-minsert-sched-nops=scheme
-mcall-sysv  -mcall-netbsd
-maix-struct-return  -msvr4-struct-return
-mabi=altivec  -mabi=no-altivec
-mabi=spe  -mabi=no-spe
-misel=yes  -misel=no
-mspe=yes  -mspe=no
-mfloat-gprs=yes  -mfloat-gprs=no
-mprototype  -mno-prototype
-msim  -mmvme  -mads  -myellowknife  -memb  -msdata
-msdata=opt  -mvxworks  -mwindiss  -G num  -pthread
```

Darwin Options

```
-all_load  -allowable_client  -arch  -arch_errors_fatal
-arch_only  -bind_at_load  -bundle  -bundle_loader
-client_name  -compatibility_version  -current_version
-dependency-file  -dylib_file  -dylinker_install_name
-dynamic  -dynamiclib  -exported_symbols_list
-filelist  -flat_namespace  -force_cpusubtype_ALL
-force_flat_namespace  -headerpad_max_install_names
-image_base  -init  -install_name  -keep_private_externs
-multi_module  -multiply_defined  -multiply_defined_unused
-noall_load  -nofixprebinding -nomultidefs  -noprebind  -noseglinkedit
-pagezero_size  -prebind  -prebind_all_twolevel_modules
-private_bundle  -read_only_relocs  -sectalign
-sectobjectsymbols  -whyload  -seg1addr
-sectcreate  -sectobjectsymbols  -sectorder
-seg_addr_table  -seg_addr_table_filename  -seglinkedit
-segprot  -segs_read_only_addr  -segs_read_write_addr
-single_module  -static  -sub_library  -sub_umbrella
-twolevel_namespace  -umbrella  -undefined
-unexported_symbols_list  -weak_reference_mismatches
-whatsloaded
```

MIPS Options

```
-EL  -EB  -march=arch  -mtune=arch
-mips1  -mips2  -mips3  -mips4  -mips32  -mips32r2  -mips64
-mips16  -mno-mips16  -mabi=abi  -mabicalls  -mno-abicalls
-mxgot  -mno-xgot  -membedded-pic  -mno-embedded-pic
-mgp32  -mgp64  -mfp32  -mfp64  -mhard-float  -msoft-float
-msingle-float  -mdouble-float  -mint64  -mlong64  -mlong32
-Gnum  -membedded-data  -mno-embedded-data
-muninit-const-in-rodata  -mno-uninit-const-in-rodata
-msplit-addresses  -mno-split-addresses
-mexplicit-relocs  -mno-explicit-relocs
-mrnames  -mno-rnames
```

```
-mcheck-zero-division  -mno-check-zero-division
-mmemcpy  -mno-memcpy  -mlong-calls  -mno-long-calls
-mmad  -mno-mad  -mfused-madd  -mno-fused-madd  -nocpp
-mfix-sb1  -mno-fix-sb1  -mflush-func=func
-mno-flush-func  -mbranch-likely  -mno-branch-likely
```

i386 and x86-64 Options

```
-mtune=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 -msse3 -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 -mno-tls-direct-seg-refs
-mcmodel=code-model
-m32  -m64
```

HPPA Options

```
-march=architecture-type
-mbig-switch  -mdisable-fpregs  -mdisable-indexing
-mfast-indirect-calls  -mgas  -mgnu-ld   -mhp-ld
-mjump-in-delay -mlinker-opt -mlong-calls
-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  -msio  -mwsio
-nolibdld  -static  -threads
```

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
-msmall-text  -mlarge-text
-mmemory-latency=time
```

DEC Alpha/VMS Options

```
-mvms-return-codes
```

H8/300 Options

```
-mrelax  -mh  -ms  -mn  -mint32  -malign-300
```

SH Options

```
-m1  -m2  -m2e  -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
-mapp-regs  -mno-app-regs
-mdisable-callt  -mno-disable-callt
-mv850e1
-mv850e
-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  -msingle-exit  -mno-single-exit
```

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-float-divide-min-latency
-minline-float-divide-max-throughput
-minline-int-divide-min-latency
-minline-int-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

```
-mtune=cpu-type  -march=cpu-type
-mhard-float  -msoft-float  -mbackchain  -mno-backchain -mkernel-backchain
-msmall-exec  -mno-small-exec  -mmvcle -mno-mvcle
-m64  -m31  -mdebug  -mno-debug  -mesa  -mzarch
-mfused-madd  -mno-fused-madd
-mwarn-framesize  -mwarn-dynamicstack  -mstack-size -mstack-guard
```

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
-mmul-bug-workaround  -mno-mul-bug-workaround
```

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

```
-mconst16 -mno-const16
-mfused-madd  -mno-fused-madd
-mtext-section-literals  -mno-text-section-literals
-mtarget-align  -mno-target-align
-mlongcalls  -mno-longcalls
```

FRV Options

```
-mgpr-32  -mgpr-64  -mfpr-32  -mfpr-64
-mhard-float  -msoft-float
-malloc-cc  -mfixed-cc  -mdword  -mno-dword
-mdouble  -mno-double
-mmedia  -mno-media  -mmuladd  -mno-muladd
-mlibrary-pic  -macc-4 -macc-8
-mpack  -mno-pack  -mno-eflags  -mcond-move  -mno-cond-move
-mscc  -mno-scc  -mcond-exec  -mno-cond-exec
-mvliw-branch  -mno-vliw-branch
-mmulti-cond-exec  -mno-multi-cond-exec  -mnested-cond-exec
-mno-nested-cond-exec  -mtomcat-stats
-mcpu=cpu
```

Code Generation Options

Section 4.18 *Options for Code Generation Conventions*.

```
-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
-fpcc-struct-return  -fpic  -fPIC -fpie -fPIE
-freg-struct-return  -fshared-data  -fshort-enums
```

```
    -fshort-double  -fshort-wchar
    -fverbose-asm  -fpack-struct  -fstack-check
    -fstack-limit-register=reg  -fstack-limit-symbol=sym
    -fargument-alias  -fargument-noalias
    -fargument-noalias-global  -fleading-underscore
    -ftls-model=model
    -ftrapv  -fwrapv  -fbounds-check
    -fvisibility
```

4.2. Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. GCC is capable of preprocessing and compiling several files either into several assembler input files, or into one assembler input file; then each assembler input file produces an object file, and 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 or C++ header file to be turned into a precompiled header.

`file.cc`
`file.cp`
`file.cxx`
`file.cpp`
`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.hh`
`file.H`

C++ header file to be turned into a precompiled header.

`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).

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

`-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.
>
> If you specify `-o` when compiling more than one input file, or you are producing an executable file as output, all the source files on the command line will be compiled at once.
>
> 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 `-Wextra` 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.

`-version`

Display the version number and copyrights of the invoked GCC.

4.3. Compiling C++ Programs

C++ source files conventionally use one of the suffixes `.C`, `.cc`, `.cpp`, `.CPP`, `.c++`, `.cp`, or `.cxx`; C++ header files often use `.hh` or `.H`; and 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 or header files from standard input, or otherwise without a suffix that flags them as C++ programs. You might also like to precompile a C header file with a `.h` extension to be used in C++ compilations. `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. Section 4.4 *Options Controlling C Dialect*, for explanations of options for languages related to C. Section 4.5 *Options Controlling C++ Dialect*, for explanations of options that are meaningful only for C++ programs.

4.4. 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 C90 programs. In C++ mode, remove GNU extensions that conflict with ISO C++.

This turns off certain features of GCC that are incompatible with ISO C90 (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`. Section 4.8 *Options to Request or Suppress Warnings*.

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. Section 6.45 *Other built-in functions provided by GCC*, for details of the functions affected.

`-std=`

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

`c89`
`iso9899:1990`

ISO C90 (same as `-ansi`).

`iso9899:199409`

ISO C90 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-3.4/c99status.html for more information. The names `c9x` and `iso9899:199x` are deprecated.

`gnu89`

Default, ISO C90 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.

`c++98`

The 1998 ISO C++ standard plus amendments.

`gnu++98`

The same as `-std=c++98` plus GNU extensions. This is the default for C++ code.

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 C90 but are in the specified version (for example, `//` comments and the `inline` keyword in ISO C99) are not disabled.

Chapter 3 *Language Standards Supported by GCC*, 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`

Don't recognize built-in functions that do not begin with `__builtin_` as prefix. Section 6.45 *Other built-in functions provided by GCC*, 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.

With the `-fno-builtin-function` option 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`.

Chapter 3 *Language Standards Supported by GCC*, for details of freestanding and hosted environments.

`-fms-extensions`

Accept some non-standard constructs used in Microsoft header files.

`-trigraphs`

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

`-no-integrated-cpp`

Performs a compilation in two passes: preprocessing and compiling. This option allows a user supplied "cc1", "cc1plus", or "cc1obj" via the `-B` option. The user supplied compilation step can then add in an additional preprocessing step after normal preprocessing but before compiling. The default is to use the integrated cpp (internal cpp)

The semantics of this option will change if "cc1", "cc1plus", and "cc1obj" are merged.

`-traditional`
`-traditional-cpp`

Formerly, these options caused GCC to attempt to emulate a pre-standard C compiler. They are now only supported with the `-E` switch. The preprocessor continues to support a pre-standard mode. 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.

`-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.

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

This option is deprecated.

4.5. 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:

`-fabi-version=n`

Use version `n` of the C++ ABI. Version 2 is the version of the C++ ABI that first appeared in G++ 3.4. Version 1 is the version of the C++ ABI that first appeared in G++ 3.2. Version 0 will always be the version that conforms most closely to the C++ ABI specification. Therefore, the ABI obtained using version 0 will change as ABI bugs are fixed.

The default is version 2.

`-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. This check is normally unnecessary because the C++ standard specifies that `operator new` will only return `0` if it is declared `throw()`, in which case the compiler will always check the return value even without this option. In all other cases, when `operator new` has a non-empty exception specification, memory exhaustion is signalled by throwing `std::bad_alloc`. 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 *`.

`-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.

`-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. Section 7.6 *Where's the Template?*, 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 some diagnostics about nonconformant code from errors to warnings. Thus, using `-fpermissive` will allow some nonconforming code to compile.

`-frepo`

Enable automatic template instantiation at link time. This option also implies `-fno-implicit-templates`. Section 7.6 *Where's the Template?*, 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.

`-fno-threadsafe-statics`

Do not emit the extra code to use the routines specified in the C++ ABI for thread-safe initialization of local statics. You can use this option to reduce code size slightly in code that doesn't need to be thread-safe.

`-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`.

`-fvisibility-inlines-hidden`

Causes all inlined methods to be marked with `__attribute__ ((visibility ("hidden")))` so that they do not appear in the export table of a DSO and do not require a

PLT indirection when used within the DSO. Enabling this option can have a dramatic effect on load and link times of a DSO as it massively reduces the size of the dynamic export table when the library makes heavy use of templates. While it can cause bloating through duplication of code within each DSO where it is used, often the wastage is less than the considerable space occupied by a long symbol name in the export table which is typical when using templates and namespaces. For even more savings, combine with the `-fvisibility=hidden` switch.

`-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. Section 4.10 *Options That Control Optimization*. Note that these functions will have linkage like inline functions; they just won't be inlined by default.

`-Wabi (C++ only)`

Warn when G++ generates code that is probably not compatible with the vendor-neutral C++ ABI. Although an effort has been made to warn about all such cases, there are probably some cases that are not warned about, even though G++ is generating incompatible code. There may also be cases where warnings are emitted even though the code that is generated will be compatible.

You should rewrite your code to avoid these warnings if you are concerned about the fact that code generated by G++ may not be binary compatible with code generated by other compilers.

The known incompatibilities at this point include:

- Incorrect handling of tail-padding for bit-fields. G++ may attempt to pack data into the same byte as a base class. For example:

  ```
  struct A { virtual void f(); int f1 : 1; };
  struct B : public A { int f2 : 1; };
  ```

  In this case, G++ will place `B::f2` into the same byte as`A::f1`; other compilers will not. You can avoid this problem by explicitly padding `A` so that its size is a multiple of the byte size on your platform; that will cause G++ and other compilers to layout `B` identically.

- Incorrect handling of tail-padding for virtual bases. G++ does not use tail padding when laying out virtual bases. For example:

  ```
  struct A { virtual void f(); char c1; };
  struct B { B(); char c2; };
  struct C : public A, public virtual B {};
  ```

  In this case, G++ will not place `B` into the tail-padding for `A`; other compilers will. You can avoid this problem by explicitly padding `A` so that its size is a multiple of its alignment (ignoring virtual base classes); that will cause G++ and other compilers to layout `C` identically.

- Incorrect handling of bit-fields with declared widths greater than that of their underlying types, when the bit-fields appear in a union. For example:

```
union U { int i : 4096; };
```

Assuming that an `int` does not have 4096 bits, G++ will make the union too small by the number of bits in an `int`.

- Empty classes can be placed at incorrect offsets. For example:

  ```
  struct A {};

  struct B {
    A a;
    virtual void f ();
  };

  struct C : public B, public A {};
  ```

  G++ will place the `A` base class of `C` at a nonzero offset; it should be placed at offset zero. G++ mistakenly believes that the `A` data member of `B` is already at offset zero.

- Names of template functions whose types involve `typename` or template template parameters can be mangled incorrectly.

  ```
  template <typename Q>
  void f(typename Q::X) {}

  template <template <typename> class Q>
  void f(typename Q<int>::X) {}
  ```

  Instantiations of these templates may be mangled incorrectly.

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

Warn when a class seems unusable because all the constructors or destructors in that class are private, and it has neither friends nor public static member functions.

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

Warn when a class appears to be polymorphic, thereby requiring a virtual destructor, yet it declares a non-virtual one. This warning is enabled by `-Wall`.

`-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) { }
};
```

The compiler will rearrange the member initializers for `i` and `j` to match the declaration order of the members, emitting a warning to that effect. This warning is enabled by `-Wall`.

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.

Also warn 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 `,`.

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

`-Wno-deprecated (C++ only)`

Do not warn about usage of deprecated features. Section 7.12 *Deprecated Features*.

`-Wno-non-template-friend (C++ only)`

Disable warnings when non-templatized friend functions are declared within a template. Since 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 search 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:

```
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 enumerated 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 =`.

4.6. Options Controlling Objective-C Dialect

(NOTE: This manual does not describe the Objective-C language itself. See http://gcc.gnu.org/readings.html for references.)

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, `-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` if the GNU runtime is being used, and `NSConstantString` if the NeXT runtime is being used (see below). The `-fconstant-cfstrings` option, if also present, will override the `-fconstant-string-class` setting and cause `@"..."` literals to be laid out as constant CoreFoundation strings.

`-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. The macro `__NEXT_RUNTIME__` is predefined if (and only if) this option is used.

`-fno-nil-receivers`

Assume that all Objective-C message dispatches (e.g., `[receiver message:arg]`) in this translation unit ensure that the receiver is not `nil`. This allows for more efficient entry points in the runtime to be used. Currently, this option is only available in conjunction with the NeXT runtime on Mac OS X 10.3 and later.

`-fobjc-exceptions`

Enable syntactic support for structured exception handling in Objective-C, similar to what is offered by C++ and Java. Currently, this option is only available in conjunction with the NeXT runtime on Mac OS X 10.3 and later.

```
@try {
  ...
     @throw expr;
  ...
}
@catch (AnObjCClass *exc) {
  ...
    @throw expr;
  ...
    @throw;
  ...
}
@catch (AnotherClass *exc) {
  ...
}
@catch (id allOthers) {
  ...
}
@finally {
  ...
    @throw expr;
  ...
}
```

The `@throw` statement may appear anywhere in an Objective-C or Objective-C++ program; when used inside of a `@catch` block, the `@throw` may appear without an argument (as shown above), in which case the object caught by the `@catch` will be rethrown.

Note that only (pointers to) Objective-C objects may be thrown and caught using this scheme. When an object is thrown, it will be caught by the nearest `@catch` clause capable of handling objects of that type, analogously to how `catch` blocks work in C++ and Java. A `@catch(id ...)` clause (as shown above) may also be provided to catch any and all Objective-C exceptions not caught by previous `@catch` clauses (if any).

The `@finally` clause, if present, will be executed upon exit from the immediately preceding `@try ... @catch` section. This will happen regardless of whether any exceptions are thrown, caught or rethrown inside the `@try ... @catch` section, analogously to the behavior of the `finally` clause in Java.

There are several caveats to using the new exception mechanism:

- Although currently designed to be binary compatible with `NS_HANDLER`-style idioms provided by the `NSException` class, the new exceptions can only be used on Mac OS X 10.3 (Panther) and later systems, due to additional functionality needed in the (NeXT) Objective-C runtime.
- As mentioned above, the new exceptions do not support handling types other than Objective-C objects. Furthermore, when used from Objective-C++, the Objective-C exception model does not interoperate with C++ exceptions at this time. This means you cannot `@throw` an exception from Objective-C and `catch` it in C++, or vice versa (i.e., `throw ... @catch`).

The `-fobjc-exceptions` switch also enables the use of synchronization blocks for thread-safe execution:

```
@synchronized (ObjCClass *guard) {
  ...
}
```

Upon entering the `@synchronized` block, a thread of execution shall first check whether a lock has been placed on the corresponding `guard` object by another thread. If it has, the current thread shall wait until the other thread relinquishes its lock. Once `guard` becomes available, the current thread will place its own lock on it, execute the code contained in the `@synchronized` block, and finally relinquish the lock (thereby making `guard` available to other threads).

Unlike Java, Objective-C does not allow for entire methods to be marked `@synchronized`. Note that throwing exceptions out of `@synchronized` blocks is allowed, and will cause the guarding object to be unlocked properly.

`-freplace-objc-classes`

Emit a special marker instructing `ld(1)` not to statically link in the resulting object file, and allow `dyld(1)` to load it in at run time instead. This is used in conjunction with the Fix-and-Continue debugging mode, where the object file in question may be recompiled and dynamically reloaded in the course of program execution, without the need to restart the program itself. Currently, Fix-and-Continue functionality is only available in conjunction with the NeXT runtime on Mac OS X 10.3 and later.

`-fzero-link`

When compiling for the NeXT runtime, the compiler ordinarily replaces calls to `objc_getClass("...")` (when the name of the class is known at compile time) with static class references that get initialized at load time, which improves run-time performance. Specifying the `-fzero-link` flag suppresses this behavior and causes calls to `objc_getClass("...")` to be retained. This is useful in Zero-Link debugging mode, since it allows for individual class implementations to be modified during program execution.

`-gen-decls`

Dump interface declarations for all classes seen in the source file to a file named `sourcename.decl`.

`-Wno-protocol`

If a class is declared to implement a protocol, a warning is issued for every method in the protocol that is not implemented by the class. The default behavior is to issue a warning for every method not explicitly implemented in the class, even if a method implementation is inherited from the superclass. If you use the `-Wno-protocol` option, then methods inherited from the superclass are considered to be implemented, and no warning is issued for them.

`-Wselector`

Warn if multiple methods of different types for the same selector are found during compilation. The check is performed on the list of methods in the final stage of compilation. Additionally, a check is performed for each selector appearing in a `@selector(...)` expression, and a corresponding method for that selector has been found during compilation. Because these checks scan the method table only at the end of compilation, these warnings are not produced if the final stage of compilation is not reached, for example because an error is found during compilation, or because the `-fsyntax-only` option is being used.

`-Wundeclared-selector`

Warn if a `@selector(...)` expression referring to an undeclared selector is found. A selector is considered undeclared if no method with that name has been declared before the

`@selector(...)` expression, either explicitly in an `@interface` or `@protocol` declaration, or implicitly in an `@implementation` section. This option always performs its checks as soon as a `@selector(...)` expression is found, while `-Wselector` only performs its checks in the final stage of compilation. This also enforces the coding style convention that methods and selectors must be declared before being used.

`-print-objc-runtime-info`

Generate C header describing the largest structure that is passed by value, if any.

4.7. 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.

4.8. 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 4.5 *Options Controlling C++ Dialect* and Section 4.6 *Options Controlling Objective-C Dialect*.

`-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. Section 6.39 *Alternate Keywords*.

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 (Section 6.25 *Declaring Attributes of Functions*), 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 C90 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). Section 4.4 *Options Controlling C Dialect*.

Since `-Wformat` also checks for null format arguments for several functions, `-Wformat` also implies `-Wnonnull`.

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

`-Wformat-y2k`

If `-Wformat` is specified, also 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.

`-Wno-format-zero-length`

If `-Wformat` is specified, do not warn about zero-length formats. The C standard specifies that zero-length formats 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 -Wformat-y2k`.

`-Wnonnull`

Warn about passing a null pointer for arguments marked as requiring a non-null value by the `nonnull` function attribute.

`-Wnonnull` is included in `-Wall` and `-Wformat`. It can be disabled with the `-Wno-nonnull` option.

`-Winit-self (C, C++, and Objective-C only)`

Warn about uninitialized variables which are initialized with themselves. Note this option can only be used with the `-Wuninitialized` option, which in turn only works with `-O1` and above.

For example, GCC will warn about `i` being uninitialized in the following snippet only when `-Winit-self` has been specified:

```
int f()
{
  int i = i;
  return i;
}
```

`-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 the GCC 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 enumerated 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.

`-Wswitch-default`

Warn whenever a `switch` statement does not have a `default` case.

`-Wswitch-enum`

Warn whenever a `switch` statement has an index of enumerated type and lacks a `case` for one or more of the named codes of that enumeration. `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 (Section 6.32 *Specifying Attributes of Variables*).

`-Wunused-parameter`

Warn whenever a function parameter is unused aside from its declaration.

To suppress this warning use the `unused` attribute (Section 6.32 *Specifying Attributes of Variables*).

`-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 (Section 6.32 *Specifying Attributes of Variables*).

`-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 the above `-Wunused` options combined.

In order to get a warning about an unused function parameter, you must either specify `-Wextra -Wunused` (note that `-Wall` implies `-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.

If you want to warn about code which uses the uninitialized value of the variable in its own initializer, use the `-Winit-self` option.

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`. Section 6.25 *Declaring Attributes of Functions*.

`-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.

`-Wstrict-aliasing`

This option is only active when `-fstrict-aliasing` is active. It warns about code which might break the strict aliasing rules that the compiler is using for optimization. The warning does not catch all cases, but does attempt to catch the more common pitfalls. It is included in `-Wall`.

`-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. This also enables some language-specific warnings described in Section 4.5 *Options Controlling C++ Dialect* and Section 4.6 *Options Controlling Objective-C Dialect*.

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.

`-Wextra`

(This option used to be called `-W`. The older name is still supported, but the newer name is more descriptive.) 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 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 };
  ```

- A function parameter is declared without a type specifier in K&R-style functions:

  ```
  void foo(bar) { }
  ```

- An empty body occurs in an `if` or `else` statement.
- A pointer is compared against integer zero with <, <=, >, or >=.
- A variable might be changed by `longjmp` or `vfork`.

- Any of several floating-point events that often indicate errors, such as overflow, underflow, loss of precision, etc.
- (C++ only) An enumerator and a non-enumerator both appear in a conditional expression.
- (C++ only) A non-static reference or non-static `const` member appears in a class without constructors.
- (C++ only) Ambiguous virtual bases.
- (C++ only) Subscripting an array which has been declared `register`.
- (C++ only) Taking the address of a variable which has been declared `register`.
- (C++ only) A base class is not initialized in a derived class' copy constructor.

`-Wno-div-by-zero`

Do not warn about compile-time integer division by zero. Floating point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs.

`-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.

`-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 analyzing 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`.
- Use of ISO C style function definitions. This warning intentionally is *not* issued for prototype declarations or variadic functions because these ISO C features will appear in your code when using libiberty's traditional C compatibility macros, `PARAMS` and `VPARAMS`. This warning is also bypassed for nested functions because that feature is already a GCC extension and thus not relevant to traditional C compatibility.

`-Wdeclaration-after-statement (C only)`

Warn when a declaration is found after a statement in a block. This construct, known from C++, was introduced with ISO C99 and is by default allowed in GCC. It is not supported by ISO C90 and was not supported by GCC versions before GCC 3.0. Section 6.24 *Mixed Declarations and Code*.

`-Wundef`

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

`-Wendif-labels`

Warn whenever an `#else` or an `#endif` are followed by text.

`-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 `-Wextra`; to get the other warnings of `-Wextra` without this warning, use `-Wextra -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.)

`-Wold-style-definition (C only)`

Warn if an old-style function definition is used. A warning is given even if there is a previous prototype.

`-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 (C only)`

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-multichar`

Do not warn if a multicharacter constant (`'FOOF'`) is used. Usually they indicate a typo in the user's code, as they have implementation-defined values, and should not be used in portable code.

`-Wno-deprecated-declarations`

Do not warn about uses of functions, variables, and types marked as deprecated by using the `deprecated` attribute. (Section 6.25 *Declaring Attributes of Functions*, Section 6.32 *Specifying Attributes of Variables*, Section 6.33 *Specifying Attributes of Types*.)

`-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 mis-aligned 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. Even with this option, the compiler will not warn about failures to inline functions declared in system headers.

The compiler uses a variety of heuristics to determine whether or not to inline a function. For example, the compiler takes into account the size of the function being inlined and the the amount of inlining that has already been done in the current function. Therefore, seemingly insignificant changes in the source program can cause the warnings produced by `-Winline` to appear or disappear.

`-Wno-invalid-offsetof (C++ only)`

Suppress warnings from applying the `offsetof` macro to a non-POD type. According to the 1998 ISO C++ standard, applying `offsetof` to a non-POD type is undefined. In existing C++ implementations, however, `offsetof` typically gives meaningful results even when applied to certain kinds of non-POD types. (Such as a simple `struct` that fails to be a POD type only by virtue of having a constructor.) This flag is for users who are aware that they are writing nonportable code and who have deliberately chosen to ignore the warning about it.

The restrictions on `offsetof` may be relaxed in a future version of the C++ standard.

`-Winvalid-pch`

Warn if a precompiled header (Section 4.20 *Using Precompiled Headers*) is found in the search path but can't be used.

`-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.

4.9. 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`, 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.

`-feliminate-unused-debug-symbols`

Produce debugging information in stabs format (if that is supported), for only symbols that are actually used.

`-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-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.

`-glevel`
`-ggdblevel`
`-gstabslevel`
`-gcofflevel`
`-gxcofflevel`
`-gvmslevel`

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 `-gdwarf-2` does not accept a concatenated debug level. Instead use an additional `-glevel` option to change the debug level for DWARF2.

`-feliminate-dwarf2-dups`

Compress DWARF2 debugging information by eliminating duplicated information about each symbol. This option only makes sense when generating DWARF2 debugging information with `-gdwarf-2`.

`-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.

`-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`

Add code so that program flow *arcs* are instrumented. During execution the program records how many times each branch and call is executed and how many times it is taken or returns. When the compiled program exits it saves this data to a file called `auxname.gcda` for each source file. The data may be used for profile-directed optimizations (`-fbranch-probabilities`), or for test coverage analysis (`-ftest-coverage`). Each object file's `auxname` is generated from the name of the output file, if explicitly specified and it is not the final executable, otherwise it is the basename of the source file. In both cases any suffix is removed (e.g. `foo.gcda` for input file `dir/foo.c`, or `dir/foo.gcda` for output file specified as `-o dir/foo.o`).

- Compile the source files with `-fprofile-arcs` plus optimization and code generation options. For test coverage analysis, use the additional `-ftest-coverage` option. You do not need to profile every source file in a program.
- Link your object files with `-lgcov` or `-fprofile-arcs` (the latter implies the former).
- Run the program on a representative workload to generate the arc profile information. This may be repeated any number of times. You can run concurrent instances of your program, and provided that the file system supports locking, the data files will be correctly updated. Also `fork` calls are detected and correctly handled (double counting will not happen).
- For profile-directed optimizations, compile the source files again with the same optimization and code generation options plus `-fbranch-probabilities` (Section 4.10 *Options That Control Optimization*).
- For test coverage analysis, use `gcov` to produce human readable information from the `.gcno` and `.gcda` files. Refer to the `gcov` documentation for further information.

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.

`-ftest-coverage`

Produce a notes file that the `gcov` code-coverage utility (Chapter 10 *`gcov`--a Test Coverage Program*) can use to show program coverage. Each source file's note file is called `auxname.gcno`. Refer to the `-fprofile-arcs` option above for a description of `auxname` and instructions on how to generate test coverage data. Coverage data will match the source files more closely, if you do not optimize.

`-dletters`

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 `dumpname`. `dumpname` is generated from the name of the output file, if explicitly specified and it is not an executable, otherwise it is the basename of the source file. In both cases any suffix is removed (e.g. `foo.01.rtl` or `foo.02.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.12.bp`.

`B`

Dump after block reordering, to `file.31.bbro`.

`c`

Dump after instruction combination, to the file `file.20.combine`.

`C`

Dump after the first if conversion, to the file `file.14.ce1`. Also dump after the second if conversion, to the file `file.21.ce2`.

`d`

Dump after branch target load optimization, to to `file.32.btl`. Also dump after delayed branch scheduling, to `file.36.dbr`.

`D`

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

`E`

Dump after the third if conversion, to `file.30.ce3`.

`f`

Dump after control and data flow analysis, to `file.11.cfg`. Also dump after life analysis, to `file.19.life`.

`F`

Dump after purging `ADDRESSOF` codes, to `file.07.addressof`.

`g`

Dump after global register allocation, to `file.25.greg`.

`G`

Dump after GCSE, to `file.08.gcse`. Also dump after jump bypassing and control flow optimizations, to `file.10.bypass`.

`h`

Dump after finalization of EH handling code, to `file.03.eh`.

`i`

Dump after sibling call optimizations, to `file.02.sibling`.

`j`

Dump after the first jump optimization, to `file.04.jump`.

`k`

Dump after conversion from registers to stack, to `file.34.stack`.

`l`

Dump after local register allocation, to `file.24.lreg`.

`L`

Dump after loop optimization passes, to `file.09.loop` and `file.16.loop2`.

`M`

Dump after performing the machine dependent reorganization pass, to `file.35.mach`.

`n`

Dump after register renumbering, to `file.29.rnreg`.

`N`

Dump after the register move pass, to `file.22.regmove`.

`o`

Dump after post-reload optimizations, to `file.26.postreload`.

`r`

Dump after RTL generation, to `file.01.rtl`.

`R`

Dump after the second scheduling pass, to `file.33.sched2`.

`s`

Dump after CSE (including the jump optimization that sometimes follows CSE), to `file.06.cse`.

`S`

Dump after the first scheduling pass, to `file.23.sched`.

`t`

Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to `file.18.cse2`.

`T`

Dump after running tracer, to `file.15.tracer`.

`u`

Dump after null pointer elimination pass to `file.05.null`.

`U`

Dump callgraph and unit-at-a-time optimization `file.00.unit`.

`V`

Dump after the value profile transformations, to `file.13.vpt`.

`w`

Dump after the second flow pass, to `file.27.flow2`.

`z`

Dump after the peephole pass, to `file.28.peephole2`.

`Z`

Dump after constructing the web, to `file.17.web`.

`a`

Produce all the dumps listed above.

`H`

Produce a core dump whenever an error occurs.

`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.01.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`.

`-frandom-seed=string`

This option provides a seed that GCC uses when it would otherwise use random numbers. It is used to generate certain symbol names that have to be different in every compiled file. It is also used to place unique stamps in coverage data files and the object files that produce them. You can use the `-frandom-seed` option to produce reproducibly identical object files.

The `string` should be different for every file you compile.

`-fsched-verbose=n`

On targets that use instruction scheduling, this option controls the amount of debugging output the scheduler prints. This information is written to standard error, unless `-dS` or `-dR` is specified, in which case it is output to the usual dump listing file, `.sched` or `.sched2` respectively. However for `n` greater than nine, the output is always printed to standard error.

For `n` greater than zero, `-fsched-verbose` outputs the same information as `-dRS`. For `n` greater than one, it also output basic block probabilities, detailed ready list information and unit/insn info. For `n` greater than two, it includes RTL at abort point, control-flow and regions info. And for `n` over four, `-fsched-verbose` also includes dependence info.

`-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 '/'. Section 4.19 *Environment Variables Affecting GCC*.

`-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.) Section 4.15 *Specifying subprocesses and the switches to pass to them.*

`-feliminate-unused-debug-types`

Normally, when producing DWARF2 output, GCC will emit debugging information for all types declared in a compilation unit, regardless of whether or not they are actually used in that compilation unit. Sometimes this is useful, such as if, in the debugger, you want to cast a value to a type that is not actually used in your program (but is declared). More often, however, this results in a significant amount of wasted space. With this option, GCC will avoid producing debug symbol output for types that are nowhere used in the source file being compiled.

4.10. Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, 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.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Using the `-funit-at-a-time` flag will allow the compiler to consider information gained from later functions in the file when compiling a function. Compiling multiple files at once to a single output file (and using `-funit-at-a-time`) will allow the compiler to use information gained from all of the files when compiling each of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed.

`-O`
`-O1`

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

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

`-O` turns on the following optimization flags:

```
-fdefer-pop
-fmerge-constants
-fthread-jumps
-floop-optimize
-fif-conversion
-fif-conversion2
-fdelayed-branch
-fguess-branch-probability
-fcprop-registers
```

`-O` also turns on `-fomit-frame-pointer` on machines where doing so does not interfere with debugging.

`-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 optimization flags specified by `-O`. It also turns on the following optimization flags:

```
-fforce-mem
-foptimize-sibling-calls
-fstrength-reduce
-fcse-follow-jumps  -fcse-skip-blocks
-frerun-cse-after-loop  -frerun-loop-opt
-fgcse  -fgcse-lm  -fgcse-sm  -fgcse-las
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fregmove
-fschedule-insns  -fschedule-insns2
-fsched-interblock  -fsched-spec
-fcaller-saves
-fpeephole2
-freorder-blocks  -freorder-functions
-fstrict-aliasing
-funit-at-a-time
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcrossjumping
```

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`, `-fweb` and `-frename-registers` options.

`-O0`

Do not optimize. This is the default.

`-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.

`-Os` disables the following optimization flags:

```
-falign-functions  -falign-jumps  -falign-loops
-falign-labels  -freorder-blocks  -fprefetch-loop-arrays
```

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 `-fflag` 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 you typically will use. You can figure out the other form by either removing `no-` or adding it.

The following options control specific optimizations. They are either activated by `-O` options or are related to ones that are. You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

`-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.

Disabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-foptimize-sibling-calls`

Optimize sibling and tail recursive calls.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

Enabled at level `-O3`.

`-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 (i.e., 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++.

Inlining is actually controlled by a number of parameters, which may be specified individually by using `-param name=value`. The `-finline-limit=n` option sets some of these parameters as follows:

`max-inline-insns-single`

is set to `n`/2.

`max-inline-insns-auto`

is set to `n`/2.

`min-inline-insns`

is set to 130 or `n`/4, whichever is smaller.

`max-inline-insns-rtl`

is set to `n`.

See below for a documentation of the individual parameters controlling inlining.

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) across compilation units.

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

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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.

`-fnew-ra`

Use a graph coloring register allocator. Currently this option is meant only for testing. Users should not specify this option, since it is not yet ready for production use.

`-fno-branch-count-reg`

Do not use "decrement and branch" instructions on a count register, but instead generate a sequence of instructions that decrement a register, compare it against zero, then branch based upon the result. This option is only meaningful on architectures that support such instructions, which include x86, PowerPC, IA-64 and S/390.

The default is `-fbranch-count-reg`, enabled when `-fstrength-reduce` is enabled.

`-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.

The default is `-ffunction-cse`

`-fno-zero-initialized-in-bss`

If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section. E.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

The default is `-fzero-initialized-in-bss`.

`-fstrength-reduce`

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

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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`.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-frerun-cse-after-loop`

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

Enabled at levels `-O2`, `-O3`, `-Os`.

`-frerun-loop-opt`

Run the loop optimizer twice.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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 elimination pass by adding `-fno-gcse` to the command line.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

Enabled by default when gcse is enabled.

`-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.

Enabled by default when gcse is enabled.

`-fgcse-las`

When `-fgcse-las` is enabled, the global common subexpression elimination pass eliminates redundant loads that come after stores to the same memory location (both partial and full redundancies).

Enabled by default when gcse is enabled.

`-floop-optimize`

Perform loop optimizations: move constant expressions out of loops, simplify exit test conditions and optionally do strength-reduction and loop unrolling as well.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-fcrossjumping`

Perform cross-jumping transformation. This transformation unifies equivalent code and save code size. The resulting code may or may not perform better than without cross-jumping.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-fif-conversion`

Attempt to transform conditional jumps into branch-less equivalents. This include use of conditional moves, min, max, set flags and abs instructions, and some tricks doable by standard arithmetics. The use of conditional execution on chips where it is available is controlled by `if-conversion2`.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-fif-conversion2`

Use conditional execution (where available) to transform conditional jumps into branch-less equivalents.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fexpensive-optimizations`

Perform a number of minor optimizations that are relatively expensive.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

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

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

Enabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-fno-sched-interblock`

Don't schedule instructions across basic blocks. This is normally enabled by default when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fno-sched-spec`

Don't allow speculative motion of non-load instructions. This is normally enabled by default when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-spec-load`

Allow speculative motion of some load instructions. This only makes sense when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-spec-load-dangerous`

Allow speculative motion of more load instructions. This only makes sense when scheduling before register allocation, i.e. with `-fschedule-insns` or at `-O2` or higher.

`-fsched-stalled-insns=n`

Define how many insns (if any) can be moved prematurely from the queue of stalled insns into the ready list, during the second scheduling pass.

`-fsched-stalled-insns-dep=n`

Define how many insn groups (cycles) will be examined for a dependency on a stalled insn that is candidate for premature removal from the queue of stalled insns. Has an effect only during the second scheduling pass, and only if `-fsched-stalled-insns` is used and its value is not zero.

`-fsched2-use-superblocks`

When scheduling after register allocation, do use superblock scheduling algorithm. Superblock scheduling allows motion across basic block boundaries resulting on faster schedules. This option is experimental, as not all machine descriptions used by GCC model the CPU closely enough to avoid unreliable results from the algorithm.

This only makes sense when scheduling after register allocation, i.e. with `-fschedule-insns2` or at `-O2` or higher.

`-fsched2-use-traces`

Use `-fsched2-use-superblocks` algorithm when scheduling after register allocation and additionally perform code duplication in order to increase the size of superblocks using tracer pass. See `-ftracer` for details on trace formation.

This mode should produce faster but significantly longer programs. Also without `-fbranch-probabilities` the traces constructed may not match the reality and hurt the performance. This only makes sense when scheduling after register allocation, i.e. with `-fschedule-insns2` or at `-O2` or higher.

`-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.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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 contact mailto:gcc@@gcc.gnu.org, and describe how use of these options affects the performance of your production code. Examples of code that runs *slower* when these options are *enabled* are very valuable.

`-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.

`-fpeephole` is enabled by default. `-fpeephole2` enabled at levels `-O2`, `-O3`, `-Os`.

`-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.

The default is `-fguess-branch-probability` at levels `-O`, `-O2`, `-O3`, `-Os`.

`-freorder-blocks`

Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality.

Enabled at levels `-O2`, `-O3`.

`-freorder-functions`

Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality. This is implemented by using special subsections `.text.hot` for most frequently executed functions and `.text.unlikely` for unlikely executed functions. Reordering is done by the linker so object file format must support named sections and linker must place them in a reasonable way.

Also profile feedback must be available in to make this option effective. See `-fprofile-arcs` for details.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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`.

Enabled at levels `-O2`, `-O3`, `-Os`.

`-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 or is zero, use a machine-dependent default.

Enabled at levels `-O2`, `-O3`.

`-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.

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

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 or is zero, use a machine-dependent default which is very likely to be `1`, meaning no alignment.

Enabled at levels `-O2`, `-O3`.

`-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.

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

If `n` is not specified or is zero, use a machine-dependent default.

Enabled at levels `-O2`, `-O3`.

`-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.

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

If `n` is not specified or is zero, use a machine-dependent default.

Enabled at levels `-O2`, `-O3`.

`-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".

`-fweb`

Constructs webs as commonly used for register allocation purposes and assign each web individual pseudo register. This allows the register allocation pass to operate on pseudos directly, but also strengthens several other optimization passes, such as CSE, loop optimizer and trivial dead code remover. It can, however, make debugging impossible, since variables will no longer stay in a "home register".

Enabled at levels `-O3`.

`-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.

Disabled at levels `-O`, `-O2`, `-O3`, `-Os`.

`-fprofile-generate`

Enable options usually used for instrumenting application to produce profile useful for later recompilation with profile feedback based optimization. You must use `-fprofile-generate` both when compiling and when linking your program.

The following options are enabled: `-fprofile-arcs`, `-fprofile-values`, `-fvpt`.

`-fprofile-use`

Enable profile feedback directed optimizations, and optimizations generally profitable only with profile feedback available.

The following options are enabled: `-fbranch-probabilities`, `-fvpt`, `-funroll-loops`, `-fpeel-loops`, `-ftracer`.

The following options control compiler behavior regarding floating point arithmetic. These options trade off between speed and correctness. All must be specifically enabled.

`-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.

`-ffast-math`

Sets `-fno-math-errno`, `-funsafe-math-optimizations`, `-fno-trapping-math`, `-ffinite-math-only`, `-fno-rounding-math` and `-fno-signaling-nans`.

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`.

`-ffinite-math-only`

Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or +-Infs.

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.

The default is `-fno-finite-math-only`.

`-fno-trapping-math`

Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation. This option implies `-fno-signaling-nans`. 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`.

`-frounding-math`

Disable transformations and optimizations that assume default floating point rounding behavior. This is round-to-zero for all floating point to integer conversions, and round-to-nearest for all other arithmetic truncations. This option should be specified for programs that change the FP rounding mode dynamically, or that may be executed with a non-default rounding mode. This option disables constant folding of floating point expressions at compile-time (which may be affected by rounding mode) and arithmetic transformations that are unsafe in the presence of sign-dependent rounding modes.

The default is `-fno-rounding-math`.

This option is experimental and does not currently guarantee to disable all GCC optimizations that are affected by rounding mode. Future versions of GCC may provide finer control of this setting using C99's `FENV_ACCESS` pragma. This command line option will be used to specify the default state for `FENV_ACCESS`.

`-fsignaling-nans`

Compile code assuming that IEEE signaling NaNs may generate user-visible traps during floating-point operations. Setting this option disables optimizations that may change the number of exceptions visible with signaling NaNs. This option implies `-ftrapping-math`.

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

The default is `-fno-signaling-nans`.

This option is experimental and does not currently guarantee to disable all GCC optimizations that affect signaling NaN behavior.

`-fsingle-precision-constant`

Treat floating point constant as single precision constant instead of implicitly converting it to double precision constant.

The following options control optimizations that may improve performance, but are not enabled by any `-O` options. This section includes experimental options that may produce broken code.

`-fbranch-probabilities`

After running a program compiled with `-fprofile-arcs` (Section 4.9 *Options for Debugging Your Program or GCC*), 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.gcda` 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_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.

`-fprofile-values`

If combined with `-fprofile-arcs`, it adds code so that some data about values of expressions in the program is gathered.

With `-fbranch-probabilities`, it reads back the data gathered from profiling values of expressions and adds `REG_VALUE_PROFILE` notes to instructions for their later usage in optimizations.

`-fvpt`

If combined with `-fprofile-arcs`, it instructs the compiler to add a code to gather information about values of expressions.

With `-fbranch-probabilities`, it reads back the data gathered and actually performs the optimizations based on them. Currently the optimizations include specialization of division operation using the knowledge about the value of the denominator.

`-fnew-ra`

Use a graph coloring register allocator. Currently this option is meant for testing, so we are interested to hear about miscompilations with `-fnew-ra`.

`-ftracer`

Perform tail duplication to enlarge superblock size. This transformation simplifies the control flow of the function allowing other optimizations to do better job.

`-funit-at-a-time`

Parse the whole compilation unit before starting to produce code. This allows some extra optimizations to take place but consumes more memory.

`-funroll-loops`

Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. `-funroll-loops` implies `-frerun-cse-after-loop`. It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations). 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`.

`-fpeel-loops`

Peels the loops for that there is enough information that they do not roll much (from profile feedback). It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations).

`-funswitch-loops`

Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

`-fold-unroll-loops`

Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop, using the old loop unroller whose loop recognition is based on notes from frontend. `-fold-unroll-loops` implies both `-fstrength-reduce` and `-frerun-cse-after-loop`. This option makes code larger, and may or may not make it run faster.

`-fold-unroll-all-loops`

Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This is done using the old loop unroller whose loop recognition is based on notes from frontend. This usually makes programs run more slowly. `-fold-unroll-all-loops` implies the same options as `-fold-unroll-loops`.

`-funswitch-loops`

Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

`-funswitch-loops`

Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

`-fprefetch-loop-arrays`

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

Disabled at level `-Os`.

`-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.

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`.

`-fbranch-target-load-optimize`

Perform branch target register load optimization before prologue / epilogue threading. The use of target registers can typically be exposed only during reload, thus hoisting loads out of loops and doing inter-block scheduling needs a separate optimization pass.

`-fbranch-target-load-optimize2`

Perform branch target register load optimization after prologue / epilogue threading.

`-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.

The names of specific parameters, and the meaning of the values, are tied to the internals of the compiler, and are subject to change without notice in future releases.

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

`max-crossjump-edges`

The maximum number of incoming edges to consider for crossjumping. The algorithm used by `-fcrossjumping` is O(N^2) in the number of edges incoming to each block. Increasing values mean more aggressive optimization, making the compile time increase with probably small improvement in executable size.

`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-single`

Several parameters control the tree inliner used in gcc. This number sets the maximum number of instructions (counted in GCC's internal representation) in a single function that the tree inliner will consider for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++). The default value is 500.

`max-inline-insns-auto`

When you use `-finline-functions` (included in `-O3`), a lot of functions that would otherwise not be considered for inlining by the compiler will be investigated. To those functions, a different (more restrictive) limit compared to functions declared inline can be applied. The default value is 100.

`large-function-insns`

The limit specifying really large functions. For functions greater than this limit inlining is constrained by `-param large-function-growth`. This parameter is useful primarily to avoid extreme compilation time caused by non-linear algorithms used by the backend. This parameter is ignored when `-funit-at-a-time` is not used. The default value is 3000.

`large-function-growth`

Specifies maximal growth of large function caused by inlining in percents. This parameter is ignored when `-funit-at-a-time` is not used. The default value is 200.

`inline-unit-growth`

Specifies maximal overall growth of the compilation unit caused by inlining. This parameter is ignored when `-funit-at-a-time` is not used. The default value is 150.

`max-inline-insns-rtl`

For languages that use the RTL inliner (this happens at a later stage than tree inlining), you can set the maximum allowable size (counted in RTL instructions) for the RTL inliner with this parameter. The default value is 600.

`max-unrolled-insns`

The maximum number of instructions that a loop should have if that loop is unrolled, and if the loop is unrolled, it determines how many times the loop code is unrolled.

`max-average-unrolled-insns`

The maximum number of instructions biased by probabilities of their execution that a loop should have if that loop is unrolled, and if the loop is unrolled, it determines how many times the loop code is unrolled.

`max-unroll-times`

The maximum number of unrollings of a single loop.

`max-peeled-insns`

The maximum number of instructions that a loop should have if that loop is peeled, and if the loop is peeled, it determines how many times the loop code is peeled.

`max-peel-times`

The maximum number of peelings of a single loop.

`max-completely-peeled-insns`

The maximum number of insns of a completely peeled loop.

`max-completely-peel-times`

The maximum number of iterations of a loop to be suitable for complete peeling.

`max-unswitch-insns`

The maximum number of insns of an unswitched loop.

`max-unswitch-level`

The maximum number of branches unswitched in a single loop.

`hot-bb-count-fraction`

Select fraction of the maximal count of repetitions of basic block in program given basic block needs to have to be considered hot.

`hot-bb-frequency-fraction`

Select fraction of the maximal frequency of executions of basic block in function given basic block needs to have to be considered hot

`tracer-dynamic-coverage`
`tracer-dynamic-coverage-feedback`

This value is used to limit superblock formation once the given percentage of executed instructions is covered. This limits unnecessary code size expansion.

The `tracer-dynamic-coverage-feedback` is used only when profile feedback is available. The real profiles (as opposed to statically estimated ones) are much less balanced allowing the threshold to be larger value.

`tracer-max-code-growth`

Stop tail duplication once code growth has reached given percentage. This is rather hokey argument, as most of the duplicates will be eliminated later in cross jumping, so it may be set to much higher values than is the desired code growth.

`tracer-min-branch-ratio`

Stop reverse growth when the reverse probability of best edge is less than this threshold (in percent).

`tracer-min-branch-ratio`
`tracer-min-branch-ratio-feedback`

Stop forward growth if the best edge do have probability lower than this threshold.

Similarly to `tracer-dynamic-coverage` two values are present, one for compilation for profile feedback and one for compilation without. The value for compilation with profile feedback needs to be more conservative (higher) in order to make tracer effective.

`max-cse-path-length`

Maximum number of basic blocks on path that cse considers.

`ggc-min-expand`

GCC uses a garbage collector to manage its own memory allocation. This parameter specifies the minimum percentage by which the garbage collector's heap should be allowed to expand between collections. Tuning this may improve compilation speed; it has no effect on code generation.

The default is 30% + 70% * (RAM/1GB) with an upper bound of 100% when RAM >= 1GB. If `getrlimit` is available, the notion of "RAM" is the smallest of actual RAM, RLIMIT_RSS, RLIMIT_DATA and RLIMIT_AS. If GCC is not able to calculate RAM on a particular platform, the lower bound of 30% is used. Setting this parameter and `ggc-min-heapsize` to zero causes a full collection to occur at every opportunity. This is extremely slow, but can be useful for debugging.

`ggc-min-heapsize`

Minimum size of the garbage collector's heap before it begins bothering to collect garbage. The first collection occurs after the heap expands by `ggc-min-expand`% beyond `ggc-min-heapsize`. Again, tuning this may improve compilation speed, and has no effect on code generation.

The default is RAM/8, with a lower bound of 4096 (four megabytes) and an upper bound of 131072 (128 megabytes). If `getrlimit` is available, the notion of "RAM" is the smallest of actual RAM, RLIMIT_RSS, RLIMIT_DATA and RLIMIT_AS. If GCC is not able to calculate RAM on a particular platform, the lower bound is used. Setting this parameter very large effectively disables garbage collection. Setting this parameter and `ggc-min-expand` to zero causes a full collection to occur at every opportunity.

`max-reload-search-insns`

The maximum number of instruction reload should look backward for equivalent register. Increasing values mean more aggressive optimization, making the compile time increase with probably slightly better performance. The default value is 100.

`max-cselib-memory-location`

The maximum number of memory locations cselib should take into acount. Increasing values mean more aggressive optimization, making the compile time increase with probably slightly better performance. The default value is 500.

`min-pretend-dynamic-size`

Force any automatic object whose size in bytes is equal to or greater than the specified value to be allocated dynamically, as if their size wasn't known to compile time. This enables their storage to be released at the end of the block containing them, reducing total stack usage if multiple functions with heavy stack use are inlined into a single function. It won't have any effect on objects that are suitable for allocation to registers (i.e., that are sufficiently small and that don't have their address taken), nor on objects allocated in the outermost block of a function. The default, zero, causes objects whose sizes are known at compile time to have storage allocated at function entry.

`reorder-blocks-duplicate`

`reorder-blocks-duplicate-feedback`

Used by basic block reordering pass to decide whether to use unconditional branch or duplicate the code on its destination. Code is duplicated when its estimated size is smaller than this value multiplied by the estimated size of unconditional jump in the hot spots of the program.

The `reorder-block-duplicate-feedback` is used only when profile feedback is available and may be set to higher values than `reorder-block-duplicate` since information about the hot spots is more accurate.

4.11. 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.

`-Xpreprocessor option`

Pass `option` as an option to the preprocessor. You can use this to supply system-specific preprocessor options which GCC does not know how to recognize.

If you want to pass an option that takes an argument, you must use `-Xpreprocessor` twice, once for the option and once for the argument.

`-D name`

Predefine `name` as a macro, with definition `1`.

`-D name=definition`

Predefine `name` as a macro, with definition `definition`. The contents of `definition` are tokenized and processed as if they appeared during translation phase three in a `#define` directive. In particular, the definition will be truncated by embedded newline characters.

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 or GCC-specific macros. The standard 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. If the directory dir is a standard system include directory, the option is ignored to ensure that the default search order for system directories and the special treatment of system headers are not defeated .

-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, -Wtrigraphs, -Wmultichar and a warning about integer promotion causing a change of sign in #if expressions. 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

Most trigraphs in comments cannot affect the meaning of the program. However, a trigraph that would form an escaped newline (??/ at the end of a line) can, by changing where the comment begins or ends. Therefore, only trigraphs that would form escaped newlines produce warnings inside a comment.

This option is implied by -Wall. If -Wall is not given, this option is still enabled unless trigraphs are enabled. To get trigraph conversion without warnings, but get the other -Wall warnings, use -trigraphs -Wall -Wno-trigraphs.

-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.

`-Wunused-macros`

Warn about macros defined in the main file that are unused. A macro is *used* if it is expanded or tested for existence at least once. The preprocessor will also warn if the macro has not been used at the time it is redefined or undefined.

Built-in macros, macros defined on the command line, and macros defined in include files are not warned about.

Note: If a macro is actually used, but only used in skipped conditional blocks, then CPP will report it as unused. To avoid the warning in such a case, you might improve the scope of the macro's definition by, for example, moving it into the first skipped block. Alternatively, you could provide a dummy use with something like:

```
#if defined the_macro_causing_the_warning
#endif
```

`-Wendif-labels`

Warn whenever an `#else` or an `#endif` are followed by text. This usually happens in code of the form

```
#if FOO
...
#else FOO
...
#endif FOO
```

The second and third `FOO` should be in comments, but often are not in older programs. This warning is on by default.

`-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` (Section 4.19 *Environment Variables Affecting GCC*). Debug output will still be sent to the regular output stream as normal.

Passing `-M` to the driver implies `-E`, and suppresses warnings with an implicit `-w`.

`-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`

In conjunction with an option such as `-M` requesting dependency generation, `-MG` assumes missing header files are generated files and adds them to the dependency list without raising an error. The dependency filename is taken directly from the `#include` directive without prepending any path. `-MG` also suppresses preprocessed output, as a missing header file renders this useless.

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 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.

`-fpch-deps`

When using precompiled headers (Section 4.20 *Using Precompiled Headers*), this flag will cause the dependency-output flags to also list the files from the precompiled header's dependencies. If not specified only the precompiled header would be listed and not the files that were used to create it because those files are not consulted when a precompiled header is used.

`-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 knows about C and C++ standards; others may 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.

`c++98`

The 1998 ISO C++ standard plus amendments.

`gnu++98`

The same as `-std=c++98` plus GNU extensions. This is the default for C++ code.

`-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.

`-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.

`-fdollars-in-identifiers`

Accept `$` in identifiers.

`-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.

`-fexec-charset=charset`

Set the execution character set, used for string and character constants. The default is UTF-8. `charset` can be any encoding supported by the system's `iconv` library routine.

`-fwide-exec-charset=charset`

Set the wide execution character set, used for wide string and character constants. The default is UTF-32 or UTF-16, whichever corresponds to the width of `wchar_t`. As with `-ftarget-charset`, `charset` can be any encoding supported by the system's `iconv` library routine; however, you will have problems with encodings that do not fit exactly in `wchar_t`.

`-finput-charset=charset`

Set the input character set, used for translation from the character set of the input file to the source character set used by GCC. If the locale does not specify, or GCC cannot get this information from the locale, the default is UTF-8. This can be overridden by either the locale or this command line option. Currently the command line option takes precedence if there's a conflict. `charset` can be any encoding supported by the system's `iconv` library routine.

`-fworking-directory`

Enable generation of linemarkers in the preprocessor output that will let the compiler know the current working directory at the time of preprocessing. When this option is enabled, the preprocessor will emit, after the initial linemarker, a second linemarker with the current working directory followed by two slashes. GCC will use this directory, when it's present in the preprocessed input, as the directory emitted as the current working directory in some debugging information formats. This option is implicitly enabled if debugging information is enabled, but this can be inhibited with the negated form `-fno-working-directory`. If the `-P` flag is present in the command line, this option has no effect, since no `#line` directives are emitted whatsoever.

`-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`.

`-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 `#`.

`-CC`

Do not discard comments, including during macro expansion. This is like `-C`, except that comments contained within macros are also passed through to the output file where the macro is expanded.

In addition to the side-effects of the `-C` option, the `-CC` option causes all C++-style comments inside a macro to be converted to C-style comments. This is to prevent later use of that macro from inadvertently commenting out the remainder of the source line.

The `-CC` option is generally used to support lint comments.

`-traditional-cpp`

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

`-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.

`-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. Precompiled header files are also printed, even if they are found to be invalid; an invalid precompiled header file is printed with `...x` and a valid one with `...!` .

`-version`
`-version`

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

4.12. 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.

`-Xassembler option`

Pass `option` as an option to the assembler. You can use this to supply system-specific assembler options which GCC does not know how to recognize.

If you want to pass an option that takes an argument, you must use `-Xassembler` twice, once for the option and once for the argument.

4.13. 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. Section 4.2 *Options Controlling the Kind of Output*.

`-llibrary`
`-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 `liblibrary.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. (, 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; .)

`-pie`

Produce a position independent executable on targets which support it. For predictable results, you must also specify the same set of options that were used to generate code (`-fpie`, `-fPIE`, or model suboptions) when you specify this option.

`-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 non-GNU linker or 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.

`-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.

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.

4.14. Options for Directory Search

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

`-Idir`

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`, the `-I` option will be ignored. The directory will still be searched but as a system directory at its normal position in the system include chain. This is to ensure that GCC's procedure to fix buggy system headers and the ordering for the include_next directive are not inadvertently changed. If you really need to change the search order for system directories, use the `-nostdinc` and/or `-isystem` options.

`-I-`

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

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

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

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

`-Ldir`

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

`-Bprefix`

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/` (Section 4.16 *Specifying Target Machine and Compiler Version*).

For each subprogram to be run, the compiler driver first tries the `-B` prefix, if any. If that name is not found, or if `-B` was not specified, the driver tries two standard prefixes, which are `/usr/lib/gcc/` and `/usr/local/lib/gcc/`. 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`. Section 4.19 *Environment Variables Affecting GCC*.

As a special kludge, if the path provided by `-B` is `[dir/]stageN/`, 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.

4.15. 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.

`%gsuffix`

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.

`%usuffix`

Like `%g`, but generates a new temporary file name even if `%usuffix` was already seen.

`%Usuffix`

Substitutes the last file name generated with `%usuffix`, generating a new one if there is no such last file name. In the absence of any `%usuffix`, this is just like `%gsuffix`, 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.

`%jsuffix`

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`
`%msuffix`

Like `%g`, except if `-pipe` is in effect. In that case `%|` substitutes a single dash and `%m` substitutes nothing at all. These are the two most common ways to instruct a program that it should read from standard input or write to standard output. If you need something more elaborate you can use an `%{pipe:X}` construct: see for example `f/lang-specs.h`.

`%.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 any of `-iprefix` (made from `GCC_EXEC_PREFIX`), `-isysroot` (made from `TARGET_SYSTEM_ROOT`), and `-isystem` (made from `COMPILER_PATH` and `-B` options) as necessary.

`%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.

`%estr`

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

`%(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`.

`%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`

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

`%:function(args)`

Call the named function `function`, passing it `args`. `args` is first processed as a nested spec string, then split into an argument vector in the usual fashion. The function returns a string which is processed as if it had appeared literally as part of the current spec.

The following built-in spec functions are provided:

`if-exists`

The `if-exists` spec function takes one argument, an absolute pathname to a file. If the file exists, `if-exists` returns the pathname. Here is a small example of its usage:

```
*startfile:
crt0%O%s %:if-exists(crti%O%s) crtbegin%O%s
```

`if-exists-else`

The `if-exists-else` spec function is similar to the `if-exists` spec function, except that it takes two arguments. The first argument is an absolute pathname to a file. If the file exists, `if-exists-else` returns the pathname. If it does not exist, it returns the second argument. This way, `if-exists-else` can be used to select one file or another, based on the existence of the first. Here is a small example of its usage:

```
*startfile:
crt0%O%s %:if-exists(crti%O%s) \
%:if-exists-else(crtbeginT%O%s crtbegin%O%s)
```

`%{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*&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:X}`

Substitutes `X`, if the `-S` switch was given to GCC.

`%{!S:X}`

Substitutes `X`, if the `-S` switch was *not* given to GCC.

`%{S*:X}`

Substitutes `X` if one or more switches whose names start with `-S` are specified to GCC. Normally `X` is substituted only once, no matter how many such switches appeared. However, if `%*` appears somewhere in `X`, then `X` will be substituted once for each matching switch, with the `%*` replaced by the part of that switch that matched the `*`.

`%{.S:X}`

Substitutes `X`, if processing a file with suffix `S`.

`%{!.S:X}`

Substitutes `X`, 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 `|`. If `%*` appears in `X`, all of the alternatives must be starred, and only the first matching alternative is substituted.

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
```

`%{S:X; T:Y; :D}`

If `S` was given to GCC, substitutes `X`; else if `T` was given to GCC, substitutes `Y`; else substitutes `D`. There can be as many clauses as you need. This may be combined with `.`, `!`, `|`, and `*` as needed.

The conditional text `X` in a %{`S:X`} or similar construct may contain other nested `%` constructs or spaces, or even newlines. They are processed as usual, as described above. Trailing white space in `X` is ignored. White space may also appear anywhere on the left side of the colon in these constructs, except between `.` or `*` and the corresponding word.

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.

4.16. Specifying Target Machine and Compiler Version

The usual way to run GCC is to run the executable called `gcc`, or <`machine`>`-gcc` when cross-compiling, or <`machine`>`-gcc-`<`version`> to run a version other than the one that was installed last. Sometimes this is inconvenient, so GCC provides options that will switch to another cross-compiler or version.

`-b machine`

The argument `machine` specifies the target machine for compilation.

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.

`-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 `-V` and `-b` options work by running the <`machine`>`-gcc-`<`version`> executable, so there's no real reason to use them if you can just run that directly.

4.17. 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.

4.17.1. 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 `401`, `403`, `405`, `405fp`, `440`, `440fp`, `505`, `601`, `602`, `603`, `603e`, `604`, `604e`, `620`, `630`, `740`, `7400`, `7450`, `750`, `801`, `821`, `823`, `860`, `970`, `common`, `ec603e`, `G3`, `G4`, `G5`, `power`, `power2`, `power3`, `power4`, `power5`, `powerpc`, `powerpc64`, `rios`, `rios1`, `rios2`, `rsc`, and `rs64a`.

`-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 the following options: `-maltivec`, `-mhard-float`, `-mmfcrf`, `-mmultiple`, `-mnew-mnemonics`, `-mpower`, `-mpower2`, `-mpowerpc64`, `-mpowerpc-gpopt`, `-mpowerpc-gfxopt`, `-mstring`. The particular options set for any particular CPU will vary between compiler versions, depending on what setting seems to produce optimal code for that CPU; it doesn't necessarily reflect the actual hardware's capabilities. If you wish to set an individual option to a particular value, you may specify it after the `-mcpu` option, like `-mcpu=970 -mno-altivec`.

On AIX, the `-maltivec` and `-mpowerpc64` options are not enabled or disabled by the `-mcpu` option at present, since AIX does not have full support for these options. You may still enable or disable them individually if you're sure it'll work in your environment.

`-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.

`-mabi=spe`

Extend the current ABI with SPE ABI extensions. This does not change the default ABI, instead it adds the SPE ABI extensions to the current ABI.

`-mabi=no-spe`

Disable Booke SPE ABI extensions for the current ABI.

`-misel=yes/no`
`-misel`

This switch enables or disables the generation of ISEL instructions.

`-mspe=yes/no`
`-mspe`

This switch enables or disables the generation of SPE simd instructions.

`-mfloat-gprs=yes/no`
`-mfloat-gprs`

This switch enables or disables the generation of floating point operations on the general purpose registers for architectures that support it. This option is currently only available on the MPC8540.

`-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.

`-malign-natural`
`-malign-power`

On Darwin and 64-bit PowerPC GNU/Linux, the option `-malign-natural` overrides the ABI-defined alignment of larger types, such as floating-point doubles, on their natural size-based boundary. The option `-malign-power` instructs GCC to follow the ABI-specified alignment rules. GCC defaults to the standard alignment defined in the ABI.

`-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`.

`-mdynamic-no-pic`

On Darwin systems, compile code so that it is not relocatable, but that its external references are relocatable. The resulting code is suitable for applications, but not shared libraries.

`-mprioritize-restricted-insns=priority`

This option controls the priority that is assigned to dispatch-slot restricted instructions during the second scheduling pass. The argument `priority` takes the value `0/1/2` to assign `no/highest/second-highest` priority to dispatch slot restricted instructions.

`-msched-costly-dep=dependence_type`

This option controls which dependences are considered costly by the target during instruction scheduling. The argument `dependence_type` takes one of the following values: `no`: no dependence is costly, `all`: all dependences are costly, `true_store_to_load`: a true dependence from store to load is costly, `store_to_load`: any dependence from store to load is costly, `number`: any dependence which latency >= `number` is costly.

`-minsert-sched-nops=scheme`

This option controls which nop insertion scheme will be used during the second scheduling pass. The argument `scheme` takes one of the following values: `no`: Don't insert nops. `pad`: Pad with nops any dispatch group which has vacant issue slots, according to the scheduler's grouping. `regroup_exact`: Insert nops to force costly dependent insns into separate groups. Insert exactly as many nops as needed to force an insn to a new group, according to the estimated processor grouping. `number`: Insert nops to force costly dependent insns into separate groups. Insert `number` nops to force an insn to a new group.

`-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-linux`

On System V.4 and embedded PowerPC systems compile code for the Linux-based GNU 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.

`-mwindiss`

Specify that you are compiling for the WindISS simulation environment.

`-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.

`-mlongcall`
`-mno-longcall`

Default to making all function calls via pointers, so that functions which reside further than 64 megabytes (67,108,864 bytes) from the current location can be called. This setting can be overridden by the `shortcall` function attribute, or by `#pragma longcall(0)`.

Some linkers are capable of detecting out-of-range calls and generating glue code on the fly. On these systems, long calls are unnecessary and generate slower code. As of this writing, the AIX linker can do this, as can the GNU linker for PowerPC/64. It is planned to add this feature to the GNU linker for 32-bit PowerPC systems as well.

On Mach-O (Darwin) systems, this option directs the compiler emit to the glue for every direct call, and the Darwin linker decides whether to use or discard it.

In the future, we may cause GCC to ignore all longcall specifications when the linker is known to generate glue.

`-pthread`

Adds support for multithreading with the *pthreads* library. This option sets flags for both the preprocessor and linker.

4.17.2. Darwin Options

These options are defined for all architectures running the Darwin operating system. They are useful for compatibility with other Mac OS compilers.

`-all_load`

Loads all members of static archive libraries. See man ld(1) for more information.

`-arch_errors_fatal`

Cause the errors having to do with files that have the wrong architecture to be fatal.

`-bind_at_load`

Causes the output file to be marked such that the dynamic linker will bind all undefined references when the file is loaded or launched.

`-bundle`

Produce a Mach-o bundle format file. See man ld(1) for more information.

`-bundle_loader executable`

This specifies the `executable` that will be loading the build output file being linked. See man ld(1) for more information.

`-allowable_client client_name`
`-arch_only`

```
-client_name
-compatibility_version
-current_version
-dependency-file
-dylib_file
-dylinker_install_name
-dynamic
-dynamiclib
-exported_symbols_list
-filelist
-flat_namespace
-force_cpusubtype_ALL
-force_flat_namespace
-headerpad_max_install_names
-image_base
-init
-install_name
-keep_private_externs
-multi_module
-multiply_defined
-multiply_defined_unused
-noall_load
-nofixprebinding
-nomultidefs
-noprebind
-noseglinkedit
-pagezero_size
-prebind
-prebind_all_twolevel_modules
-private_bundle
-read_only_relocs
-sectalign
-sectobjectsymbols
-whyload
-seg1addr
-sectcreate
-sectobjectsymbols
-sectorder
-seg_addr_table
-seg_addr_table_filename
-seglinkedit
-segprot
-segs_read_only_addr
-segs_read_write_addr
-single_module
-static
-sub_library
-sub_umbrella
-twolevel_namespace
-umbrella
-undefined
-unexported_symbols_list
-weak_reference_mismatches
-whatsloaded
```

These options are available for Darwin linker. Darwin linker man page describes them in detail.

4.17.3. Intel 386 and AMD x86-64 Options

These `-m` options are defined for the i386 and x86-64 family of computers:

`-mtune=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

Original Intel's i386 CPU.

i486

Intel's i486 CPU. (No scheduling is implemented for this chip.)

i586, pentium

Intel Pentium CPU with no MMX support.

pentium-mmx

Intel PentiumMMX CPU based on Pentium core with MMX instruction set support.

i686, pentiumpro

Intel PentiumPro CPU.

pentium2

Intel Pentium2 CPU based on PentiumPro core with MMX instruction set support.

pentium3, pentium3m

Intel Pentium3 CPU based on PentiumPro core with MMX and SSE instruction set support.

pentium-m

Low power version of Intel Pentium3 CPU with MMX, SSE and SSE2 instruction set support. Used by Centrino notebooks.

pentium4, pentium4m

Intel Pentium4 CPU with MMX, SSE and SSE2 instruction set support.

prescott

Improved version of Intel Pentium4 CPU with MMX, SSE, SSE2 and SSE3 instruction set support.

nocona

Improved version of Intel Pentium4 CPU with 64-bit extensions, MMX, SSE, SSE2 and SSE3 instruction set support.

k6

AMD K6 CPU with MMX instruction set support.

k6-2, k6-3

Improved versions of AMD K6 CPU with MMX and 3dNOW! instruction set support.

athlon, athlon-tbird

AMD Athlon CPU with MMX, 3dNOW!, enhanced 3dNOW! and SSE prefetch instructions support.

athlon-4, athlon-xp, athlon-mp

Improved AMD Athlon CPU with MMX, 3dNOW!, enhanced 3dNOW! and full SSE instruction set support.

k8, opteron, athlon64, athlon-fx

AMD K8 core based CPUs with x86-64 instruction set support. (This supersets MMX, SSE, SSE2, 3dNOW!, enhanced 3dNOW! and 64-bit instruction set extensions.)

winchip-c6

IDT Winchip C6 CPU, dealt in same way as i486 with additional MMX instruction set support.

winchip2

IDT Winchip2 CPU, dealt in same way as i486 with additional MMX and 3dNOW! instruction set support.

c3

Via C3 CPU with MMX and 3dNOW! instruction set support. (No scheduling is implemented for this chip.)

c3-2

Via C3-2 CPU with MMX and SSE instruction set support. (No scheduling is implemented for this chip.)

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.

`-march=cpu-type`

Generate instructions for the machine type `cpu-type`. The choices for `cpu-type` are the same as for `-mtune`. Moreover, specifying `-march=cpu-type` implies `-mtune=cpu-type`.

`-mcpu=cpu-type`

A deprecated synonym for `-mtune`.

```
-m386
-m486
-mpentium
-mpentiumpro
```

These options are synonyms for `-mtune=i386`, `-mtune=i486`, `-mtune=pentium`, and `-mtune=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 precision 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 the 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 the x86-64 compiler.

`sse,387`

Attempt to utilize both instruction sets at once. This effectively 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 the 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 and will not be binary compatible with structures in code compiled without that switch.

`-m96bit-long-double`
`-m128bit-long-double`

These switches control the size of `long double` type. The i386 application binary interface specifies the size to be 96 bits, so `-m96bit-long-double` is the default in 32 bit mode.

Modern architectures (Pentium and newer) would prefer `long double` to be aligned to an 8 or 16 byte boundary. In arrays or structures conforming to the ABI, this would not be possible. So specifying a `-m128bit-long-double` will align `long double` to a 16 byte boundary by padding the `long double` with an additional 32 bit zero.

In the x86-64 compiler, `-m128bit-long-double` is the default choice as its ABI specifies that `long double` is to be aligned on 16 byte boundary.

Notice that neither of these options enable any extra precision over the x87 standard of 80 bits for a `long double`.

Warning: if you override the default value for your target ABI, the structures and arrays containing `long double` variables will change their size as well as function calling convention for function taking `long double` will be modified. Hence they will not be binary compatible with arrays or structures in code compiled without that switch.

`-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`. Section 6.25 *Declaring Attributes of Functions*.

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

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

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

`-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`. Section 6.25 *Declaring Attributes of Functions*.

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

`-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`
`-msse3`
`-mno-sse3`
`-m3dnow`
`-mno-3dnow`

These switches enable or disable the use of built-in functions that allow direct access to the MMX, SSE, SSE2, SSE3 and 3Dnow extensions of the instruction set.

Section 6.46.1 *X86 Built-in Functions*, for details of the functions enabled and disabled by these switches.

To have SSE/SSE2 instructions generated automatically from floating-point code, see `-mfpmath=sse`.

`-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 doesn'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.

`-mtls-direct-seg-refs`
`-mno-tls-direct-seg-refs`

Controls whether TLS variables may be accessed with offsets from the TLS segment register (`%gs` for 32-bit, `%fs` for 64-bit), or whether the thread base pointer must be added. Whether or not this is legal depends on the operating system, and whether it maps the segment to cover the entire TLS area.

For systems that use GNU libc, the default is on.

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.

`-mcmodel=small`

Generate code for the small code model: the program and its symbols must be linked in the lower 2 GB of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked. This is the default code model.

`-mcmodel=kernel`

Generate code for the kernel code model. The kernel runs in the negative 2 GB of the address space. This model has to be used for Linux kernel code.

`-mcmodel=medium`

Generate code for the medium model: The program is linked in the lower 2 GB of the address space but symbols can be located anywhere in the address space. Programs can be statically or dynamically linked, but building of shared libraries are not supported with the medium model.

`-mcmodel=large`

Generate code for the large model: This model makes no assumptions about addresses and sizes of sections. Currently GCC does not implement this model.

4.17.4. 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 HP-UX.

`-mlittle-endian`

Generate code for a little endian target. This is the default for AIX5 and GNU/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-float-divide-min-latency`

Generate code for inline divides of floating point values using the minimum latency algorithm.

`-minline-float-divide-max-throughput`

Generate code for inline divides of floating point values using the maximum throughput algorithm.

`-minline-int-divide-min-latency`

Generate code for inline divides of integer values using the minimum latency algorithm.

`-minline-int-divide-max-throughput`

Generate code for inline divides of integer values 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.

`-mearly-stop-bits`
`-mno-early-stop-bits`

Allow stop bits to be placed earlier than immediately preceding the instruction that triggered the stop bit. This can improve instruction scheduling, but does not always do so.

4.17.5. 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`
`-mkernel-backchain`

In order to provide a backchain the address of the caller's frame is stored within the callee's stack frame. A backchain may be needed to allow debugging using tools that do not understand DWARF-2 call frame information. For `-mno-backchain` no backchain is maintained at all which is the default. If one of the other options is present the backchain pointer is placed either on top of the stack frame (`-mkernel-backchain`) or on the bottom (`-mbackchain`). Beside the different backchain location `-mkernel-backchain` also changes stack frame layout breaking the ABI. This option is intended to be used for code which internally needs a backchain but has to get by with a limited stack size e.g. the linux kernel. Internal unwinding code not using DWARF-2 info has to be able to locate the return address of a function. That will be eased be the fact that the return address of a function is placed two words below the backchain pointer.

`-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 GNU/Linux for S/390 ABI. When `-m64` is specified, generate code compliant to the GNU/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`.

`-mzarch`
`-mesa`

When `-mzarch` is specified, generate code using the instructions available on z/Architecture. When `-mesa` is specified, generate code using the instructions available on ESA/390. Note that `-mesa` is not possible with `-m64`. When generating code compliant to the GNU/Linux for S/390 ABI, the default is `-mesa`. When generating code compliant to the GNU/Linux for zSeries ABI, the default is `-mzarch`.

`-mmvcle`
`-mno-mvcle`

Generate (or do not generate) code using the `mvcle` instruction to perform block moves. When `-mno-mvcle` is specified, 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.

`-march=cpu-type`

Generate code that will run on `cpu-type`, which is the name of a system representing a certain processor type. Possible values for `cpu-type` are `g5`, `g6`, `z900`, and `z990`. When generating code using the instructions available on z/Architecture, the default is `-march=z900`. Otherwise, the default is `-march=g5`.

`-mtune=cpu-type`

Tune to `cpu-type` everything applicable about the generated code, except for the ABI and the set of available instructions. The list of `cpu-type` values is the same as for `-march`. The default is the value used for `-march`.

`-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 point is used.

`-mwarn-framesize=framesize`

Emit a warning if the current function exceeds the given frame size. Because this is a compile time check it doesn't need to be a real problem when the program runs. It is intended to identify functions which most probably cause a stack overflow. It is useful to be used in an environment with limited stack size e.g. the linux kernel.

`-mwarn-dynamicstack`

Emit a warning if the function calls alloca or uses dynamically sized arrays. This is generally a bad idea with a limited stack size.

`-mstack-guard=stack-guard`
`-mstack-size=stack-size`

These arguments always have to be used in conjunction. If they are present the s390 back end emits additional instructions in the function prologue which trigger a trap if the stack size is `stack-guard` bytes above the `stack-size` (remember that the stack on s390 grows downward). These options are intended to be used to help debugging stack overflow problems. The addtionally emitted code cause only little overhead and hence can also be used in production like systems without greater performance degradation. The given values have to be exact powers of 2 and `stack-size` has to be greater than `stack-guard`. In order to be effecient the extra code makes the assumption that the stack starts at an address aligned to the value given by `stack-size`. So don't expect this to work correctly with a 8k stack size and an initial stack pointer like 0xffffefff.

4.18. 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.

`-fbounds-check`

For front-ends that support it, generate additional code to check that indices used to access arrays are within the declared range. This is currently only supported by the Java and Fortran 77 front-ends, where this option defaults to true and false respectively.

`-ftrapv`

This option generates traps for signed overflow on addition, subtraction, multiplication operations.

`-fwrapv`

This option instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. This flag enables some optimizations and disables other. This option is enabled by default for the Java front-end, as required by the Java language specification.

`-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, particularly the Portable C Compiler (pcc).

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.

Warning: code compiled with the `-fpcc-struct-return` switch is not binary compatible with code compiled with the `-freg-struct-return` switch. Use it to conform to a non-default application binary interface.

`-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.

Warning: code compiled with the `-freg-struct-return` switch is not binary compatible with code compiled with the `-fpcc-struct-return` switch. Use it to conform to a non-default application binary interface.

`-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.

Warning: the `-fshort-enums` switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-fshort-double`

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

Warning: the `-fshort-double` switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-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.

Warning: the `-fshort-wchar` switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-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.

`-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.

`-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 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 and the SPARC.

Position-independent code requires special support, and therefore works only on certain machines.

`-fpie`
`-fPIE`

These options are similar to `-fpic` and `-fPIC`, but generated position independent code can be only linked into executables. Usually these options are used when `-pie` GCC option will be used during linking.

`-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.

Warning: the `-fpack-struct` switch causes GCC to generate code that is not binary compatible with code generated without that switch. Additionally, it makes the code suboptimal. Use it to conform to a non-default application binary interface.

`-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 currently disables function inlining. This restriction is expected to be removed in future releases.

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.

Warning: the `-fleading-underscore` switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface. Not all targets provide complete support for this switch.

`-ftls-model=model`

Alter the thread-local storage model to be used (Section 6.49 *Thread-Local Storage*). The `model` argument should be one of `global-dynamic`, `local-dynamic`, `initial-exec` or `local-exec`.

The default without `-fpic` is `initial-exec`; with `-fpic` the default is `global-dynamic`.

`-fvisibility=default|internal|hidden|protected`

Set the default ELF image symbol visibility to the specified option - all symbols will be marked with this unless overrided within the code. Using this feature can very substantially improve linking and load times of shared object libraries, produce more optimised code, provide near-perfect API export and prevent symbol clashes. It is *strongly* recommended that you use this in any shared objects you distribute.

Despite the nomenclature, `default` always means public ie; available to be linked against from outside the shared object. `protected` and `internal` are pretty useless in real-world usage so the only other commonly used option will be `hidden`. The default if -fvisibility isn't specified is `default` ie; make every symbol public - this causes the same behaviour as previous versions of GCC.

A good explanation of the benefits offered by ensuring ELF symbols have the correct visibility is given by "How To Write Shared Libraries" by Ulrich Drepper (which can be found at http://people.redhat.com/~drepper/) - however a superior solution made possible by this option to marking things hidden when the default is public is to make the default hidden and mark things public. This is the norm with DLL's on Windows and with `-fvisibility=hidden` and `__attribute__ ((visibility("default")))` instead of `__declspec(dllexport)` you get almost identical semantics with identical syntax. This is a great boon to those working with cross-platform projects.

For those adding visibility support to existing code, you may find `#pragma GCC visibility` of use. This works by you enclosing the declarations you wish to set visibility for with (for example) `#pragma GCC visibility push(hidden)` and `#pragma GCC visibility pop`. These can be nested up to sixteen times. Bear in mind that symbol visibility should be viewed *as part of the API interface contract* and thus all new code should always specify visibility when it is not the default ie; declarations only for use within the local DSO should *always* be marked explicitly as hidden as so to avoid PLT indirection overheads - making this abundantly clear also aids readability and self-documentation of the code. Note that due to ISO C++ specification requirements, operator new and operator delete must always be of default visibility.

An overview of these techniques, their benefits and how to use them is at http://www.nedprod.com/programs/gccvisibility.html.

4.19. 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` (Section 4.14 *Options for Directory Search*). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GCC. .

`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_GB.UTF-8` for English in the United Kingdom encoded in UTF-8.

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/` 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` (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 Microsoft 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. This 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.

In all these variables, an empty element instructs the compiler to search its current working directory. Empty elements can appear at the beginning or end of a path. For instance, if the value of `CPATH` is `:/special/include`, that has the same effect as `-I. -I/special/include`.

`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` (Section 4.11 *Options Controlling the Preprocessor*), with an optional `-MT` switch too.

`SUNPRO_DEPENDENCIES`

This variable is the same as `DEPENDENCIES_OUTPUT` (see above), except that system header files are not ignored, so it implies `-M` rather than `-MM`. However, the dependence on the main input file is omitted. Section 4.11 *Options Controlling the Preprocessor*.

4.20. Using Precompiled Headers

Often large projects have many header files that are included in every source file. The time the compiler takes to process these header files over and over again can account for nearly all of the time required to build the project. To make builds faster, GCC allows users to 'precompile' a header file; then, if builds can use the precompiled header file they will be much faster.

Caution: There are a few known situations where GCC will crash when trying to use a precompiled header. If you have trouble with a precompiled header, you should remove the precompiled header and compile without it. In addition, please use GCC's on-line defect-tracking system to report any problems you encounter with precompiled headers.

To create a precompiled header file, simply compile it as you would any other file, if necessary using the `-x` option to make the driver treat it as a C or C++ header file. You will probably want to use a tool like `make` to keep the precompiled header up-to-date when the headers it contains change.

A precompiled header file will be searched for when `#include` is seen in the compilation. As it searches for the included file () the compiler looks for a precompiled header in each directory just before it looks for the include file in that directory. The name searched for is the name specified in the `#include` with `.gch` appended. If the precompiled header file can't be used, it is ignored.

For instance, if you have `#include "all.h"`, and you have `all.h.gch` in the same directory as `all.h`, then the precompiled header file will be used if possible, and the original header will be used otherwise.

Alternatively, you might decide to put the precompiled header file in a directory and use `-I` to ensure that directory is searched before (or instead of) the directory containing the original header. Then, if you want to check that the precompiled header file is always used, you can put a file of the same name as the original header in this directory containing an `#error` command.

This also works with `-include`. So yet another way to use precompiled headers, good for projects not designed with precompiled header files in mind, is to simply take most of the header files used by a project, include them from another header file, precompile that header file, and `-include` the precompiled header. If the header files have guards against multiple inclusion, they will be skipped because they've already been included (in the precompiled header).

If you need to precompile the same header file for different languages, targets, or compiler options, you can instead make a *directory* named like `all.h.gch`, and put each precompiled header in the directory. (It doesn't matter what you call the files in the directory, every precompiled header in the directory will be considered.) The first precompiled header encountered in the directory that is valid for this compilation will be used; they're searched in no particular order.

There are many other possibilities, limited only by your imagination, good sense, and the constraints of your build system.

A precompiled header file can be used only when these conditions apply:

- Only one precompiled header can be used in a particular compilation.
- A precompiled header can't be used once the first C token is seen. You can have preprocessor directives before a precompiled header; you can even include a precompiled header from inside another header, so long as there are no C tokens before the `#include`.
- The precompiled header file must be produced for the same language as the current compilation. You can't use a C precompiled header for a C++ compilation.
- The precompiled header file must be produced by the same compiler version and configuration as the current compilation is using. The easiest way to guarantee this is to use the same compiler binary for creating and using precompiled headers.
- Any macros defined before the precompiled header (including with `-D`) must either be defined in the same way as when the precompiled header was generated, or must not affect the precompiled header, which usually means that the they don't appear in the precompiled header at all.
- Certain command-line options must be defined in the same way as when the precompiled header was generated. At present, it's not clear which options are safe to change and which are not; the safest choice is to use exactly the same options when generating and using the precompiled header.

For all of these but the last, the compiler will automatically ignore the precompiled header if the conditions aren't met. For the last item, some option changes will cause the precompiled header to be rejected, but not all incompatible option combinations have yet been found.

4.21. 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.

Section 11.10 *Caveats of using `protoize`*, for more information on how to use `protoize` successfully.

Chapter 5. 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.

5.1. Translation

- [How a diagnostic is identified (3.10, 5.1.1.3).]

  Diagnostics consist of all the output sent to stderr by GCC.
- [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).]

5.2. Environment

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

5.3. 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).]

  For internal names, all characters are significant. For external names, the number of significant characters are defined by the linker; for almost all targets, all characters are significant.

5.4. 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).]

5.5. 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).]

  GCC supports only two's complement integer types, and all bit patterns are ordinary values.
- [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).]

5.6. 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).]

5.7. 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).]

5.8. Hints

- [The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).]

  The `register` specifier affects code generation only in these ways:

  - When used as part of the register variable extension, see Section 6.38 *Variables in Specified Registers*.
  - When `-O0` is in use, the compiler allocates distinct stack memory for all variables that do not have the `register` storage-class specifier; if `register` is specified, the variable may have a shorter lifespan than the code would indicate and may never be placed in memory.
  - On some rare x86 targets, `setjmp` doesn't save the registers in all circumstances. In those cases, GCC doesn't allocate any variables in registers unless they are marked `register`.

- [The extent to which suggestions made by using the inline function specifier are effective (6.7.4).]

  GCC will not inline any functions if the `-fno-inline` option is used or if `-O0` is used. Otherwise, GCC may still be unable to inline a function for many reasons; the `-Winline` option may be used to determine if a function has not been inlined and why not.

5.9. 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).]

1. Future versions of GCC may zero-extend, or use a target-defined `ptr_extend` pattern. Do not rely on sign extension.

- [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).]

5.10. Qualifiers

- [What constitutes an access to an object that has volatile-qualified type (6.7.3).]

5.11. 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).]
- [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).]

  GCC imposes a limit of 200 nested `#include`s.
- [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).]

  If the date and time are not available, `__DATE__` expands to `"??? ?? ????"` and `__TIME__` expands to `"??:??:??"`.

5.12. Library functions

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

5.13. 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).]

5.14. Locale-specific behavior

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

Chapter 6. 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++. Chapter 7 *Extensions to the C++ Language*, 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++.

6.1. 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 taken as `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` (Section 6.6 *Referring to a Type with `typeof`*).

In G++, the result value of a statement expression undergoes array and function pointer decay, and is returned by value to the enclosing expression. For instance, if `A` is a class, then

```
        A a;

        ({a;}).Foo ()
```

will construct a temporary `A` object to hold the result of the statement expression, and that will be used to invoke `Foo`. Therefore the `this` pointer observed by `Foo` will not be the address of `a`.

Any temporaries created within a statement within a statement expression will be destroyed at the statement's end. This makes statement expressions inside macros slightly different from function calls. In the latter case temporaries introduced during argument evaluation will be destroyed at the end of the statement that includes the function call. In the statement expression case they will be destroyed during the statement expression. For instance,

```
#define macro(a)  ({__typeof__(a) b = (a); b + 3; })
template<typename T> T function(T a) { T b = a; return b + 3; }

void foo ()
{
  macro (X ());
  function (X ());
}
```

will have different places where temporaries are destroyed. For the `macro` case, the temporary `X` will be destroyed just after the initialization of `b`. In the `function` case that temporary will be destroyed when the function returns.

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.)

6.2. Locally Declared Labels

GCC allows you to declare *local labels* in any nested block scope. A local label is just like an ordinary label, but you can only reference it (with a `goto` statement, or by taking its address) within the block in which it was declared.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, /* ... */;
```

Local label declarations must come at the beginning of the block, before any ordinary declarations or statements.

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 for complex macros. If a 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(value, array, target)              \
do {                                              \
  __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:;                                          \
} while (0)
```

This could also be written using a statement-expression:

```
#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;                                          \
})
```

Local label declarations also make the labels they declare visible to nested functions, if there are any. Section 6.4 *Nested Functions*, for details.

6.3. 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.

6.4. 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);
```

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.

```
}
```

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/~aaronl/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 (Section 6.2 *Locally Declared Labels*). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well. Here is an example:

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

 /* Control comes here from access
    if it detects an error.  */
 failure:
  return -1;
}
```

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

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

6.5. 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).

However, these built-in functions may interact badly with some sophisticated features or other extensions of the language. It is, therefore, not recommended to use them outside very simple functions acting as mere forwarders for their arguments.

void *`__builtin_apply_args` () 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) 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) 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.

6.6. 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. Section 6.39 *Alternate Keywords*.

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

typeof is often 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 and evaluates each of its arguments exactly once:

```
#define max(a,b) \
  ({ typeof (a) _a = (a); \
      typeof (b) _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.

Some more examples of the use of `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, 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`.

Compatibility Note: In addition to `typeof`, GCC 2 supported a more limited extension which permitted one to write

```
typedef T = expr;
```

with the effect of declaring `T` to have the type of the expression `expr`. This extension does not work with GCC 3 (versions between 3.0 and 3.2 will crash; 3.2.1 and later give an error). Code which relies on it should be rewritten to use `typeof`:

```
typedef typeof(expr) T;
```

This will work with all versions of GCC.

6.7. 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. All these extensions are deprecated.

Standard C++ allows compound expressions and conditional expressions as lvalues, and permits casts to reference type, so use of this extension is not supported 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. This extension is deprecated. 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`.

6.8. 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.

6.9. 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.

6.10. 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). Only the DWARF2 debug info format can represent this, so use of DWARF2 is recommended. If you are using the stabs debug info format, 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.

6.11. 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 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`.

6.12. 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 C90, 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.
- A structure containing a flexible array member, or a union containing such a structure (possibly recursively), may not be a member of a structure or an element of an array. (However, these uses are permitted by GCC as extensions.)

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.
```

6.13. Structures With No Members

GCC permits a C structure to have no members:

```
struct empty {
};
```

The structure will have size zero. In C++, empty structures are part of the language. G++ treats empty structures as if they had a single member of type `char`.

6.14. 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.

6.15. 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.

6.16. Slightly Looser Rules for Escaped Newlines

Recently, the 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, 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.

6.17. 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];
}
```

6.18. 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.

6.19. 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 };
  /* ... */
}
```

6.20. 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};
```

6.21. 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. (Section 6.23 *Cast to a Union Type*.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a 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.

6.22. 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:
```

6.23. 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. (Section 6.20 *Compound Literals*.)

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);
```

6.24. 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.

6.25. 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`, `nothrow`, `format`, `format_arg`, `no_instrument_function`, `section`, `constructor`, `destructor`, `used`, `unused`, `deprecated`, `weak`, `malloc`, `alias`, `warn_unused_result` and `nonnull`. Several other attributes are defined for functions on particular target systems. Other attributes, including `section` are supported for variables declarations (Section 6.32 *Specifying Attributes of Variables*) and for types (Section 6.33 *Specifying Attributes of Types*).

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

Section 6.26 *Attribute Syntax*, 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.

The `noreturn` keyword does not affect the exceptional path when that applies: a `noreturn`-marked function may still return to the caller by throwing an exception.

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.

`nothrow`

The `nothrow` attribute is used to inform the compiler that a function cannot throw an exception. For example, most functions in the standard C library can be guaranteed not to throw an exception with the notable exceptions of `qsort` and `bsearch` that take function pointer arguments. The `nothrow` attribute is not implemented in GCC versions earlier than 3.2.

`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. Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two, not one, when giving values for `string-index` and `first-to-check`.

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`. Section 4.4 *Options Controlling C Dialect*.

`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 one). Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two.

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. Section 4.4 *Options Controlling C Dialect*.

`nonnull (arg-index, ...)`

The `nonnull` attribute specifies that some function parameters should be non-null pointers. For instance, the declaration:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
 __attribute__((nonnull (1, 2)));
```

causes the compiler to check that, in calls to `my_memcpy`, arguments `dest` and `src` are non-null. If the compiler determines that a null pointer is passed in an argument slot marked as non-null, and the `-Wnonnull` option is enabled, a warning is issued. The compiler may also choose to make optimizations based on the knowledge that certain function arguments will not be null.

If no argument index list is given to the `nonnull` attribute, all pointer arguments are marked as non-null. To illustrate, the following declaration is equivalent to the previous example:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
 __attribute__((nonnull));
```

`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.

`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 (Section 6.32 *Specifying Attributes of Variables*, Section 6.33 *Specifying Attributes of Types*.)

`warn_unused_result`

The `warn_unused_result` attribute causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as `realloc`.

```
int fn () __attribute__ ((warn_unused_result));
int foo ()
{
  if (fn () < 0) return -1;
  fn ();
  return 0;
}
```

results in warning on line 5.

`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 any non-`NULL` pointer it returns cannot alias any other pointer valid when the function returns. This will often improve optimization. Standard functions with this property include `malloc` and `calloc`. `realloc`-like functions have this property as long as the old pointer is never referred to (including comparing it to the new pointer) after the function returns a non-`NULL` value.

`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.

`visibility ("visibility_type")`

The `visibility` attribute on ELF targets causes the declaration to be emitted with default, hidden, protected or internal visibility.

```
void __attribute__ ((visibility ("protected")))
f () { /* Do something. */; }
int i __attribute__ ((visibility ("hidden")));
```

See the ELF gABI for complete details, but the short story is:

default

Default visibility is the normal case for ELF. This value is available for the visibility attribute to override other options that may change the assumed visibility of symbols.

hidden

Hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so no other *module* (executable or shared library) can reference it directly.

protected

Protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind to the local symbol. That is, the symbol cannot be overridden by another module.

internal

Internal visibility is like hidden visibility, but with additional processor specific semantics. Unless otherwise specified by the psABI, GCC defines internal visibility to mean that the function is *never* called from another module. Note that hidden symbols, while they cannot be referenced directly by other modules, can be referenced indirectly via function pointers. By indicating that a symbol cannot be called from outside the module, GCC may for instance omit the load of a PIC register since it is known that the calling function loaded the correct value.

Not all ELF targets 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.

Beware that on some ELF systems this attribute is unsuitable for global functions in shared libraries with lazy binding (which is the default). Lazy binding will send the first call via resolving code in the loader, which might assume EAX, EDX and ECX can be clobbered, as per the standard calling conventions. GNU systems with GLIBC 2.1 or higher, and FreeBSD, are believed to be safe since the loaders there save all registers. (Lazy binding can be disabled with the linker or the loader if desired, to avoid the problem.)

`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.

`fastcall`

On the Intel 386, the `fastcall` attribute causes the compiler to pass the first two arguments in the registers ECX and EDX. Subsequent arguments are passed on the stack. The called function will pop the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

`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.

`longcall/shortcall`

On the RS/6000 and PowerPC, the `longcall` attribute causes the compiler to always call this function via a pointer, just as it would if the `-mlongcall` option had been specified. The `shortcall` attribute causes the compiler not to do this. These attributes override both the `-mlongcall` switch and the `#pragma longcall` setting.

Section 4.17.1 *IBM RS/6000 and PowerPC Options*, for more information on whether long calls are necessary.

`long_call/short_call`

This attribute specifies how a particular function is called on ARM. Both attributes override the `-mlong-calls` 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.

`function_vector`

Use this attribute on the H8/300, H8/300H, and H8S 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 H8S) 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, C4x, 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 m68k, H8/300, H8/300H, H8S, 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 m68k, H8/300, H8/300H, H8S, 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_handler` 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, H8/300H, and H8S 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 and H8S 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.

`saveall`

Use this attribute on the H8/300, H8/300H, and H8S to indicate that all registers except the stack pointer should be saved in the prologue regardless of whether they are used or not.

`signal`

Use this attribute on the AVR to indicate that the specified function is a signal handler. The compiler will generate function entry and exit sequences suitable for use in a signal handler when this attribute is present. Interrupts will be disabled inside the function.

`naked`

Use this attribute on the ARM, AVR, C4x and IP2K ports to indicate that the specified function does not need prologue/epilogue sequences generated by the compiler. It is up to the programmer to provide these sequences.

`model (model-name)`

On the M32R/D, use this attribute to set the addressability of an object, and of 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).

On IA-64, use this attribute to set the addressability of an object. At present, the only supported identifier for `model-name` is `small`, indicating addressability via "small" (22-bit) addresses (so that their addresses can be loaded with the `addl` instruction). Caveat: such addressing is by definition not position independent and hence this attribute must not be used for objects defined by shared libraries.

`far`

On 68HC11 and 68HC12 the `far` attribute causes the compiler to use a calling convention that takes care of switching memory banks when entering and leaving a function. This calling convention is also the default when using the `-mlong-calls` option.

On 68HC12 the compiler will use the `call` and `rtc` instructions to call and return from a function.

On 68HC11 the compiler will generate a sequence of instructions to invoke a board-specific routine to switch the memory bank and call the real function. The board-specific routine simulates a `call`. At the end of a function, it will jump to a board-specific routine instead of using `rts`. The board-specific return routine simulates the `rtc`.

`near`

On 68HC11 and 68HC12 the `near` attribute causes the compiler to use the normal calling convention based on `jsr` and `rts`. This attribute can be used to cancel the effect of the `-mlong-calls` option.

`dllimport`

On Microsoft Windows targets, the `dllimport` attribute causes the compiler to reference a function or variable via a global pointer to a pointer that is set up by the Microsoft Windows dll library. The pointer name is formed by combining `_imp__` and the function or variable name. The attribute implies `extern` storage.

Currently, the attribute is ignored for inlined functions. If the attribute is applied to a symbol *definition*, an error is reported. If a symbol previously declared `dllimport` is later defined, the attribute is ignored in subsequent references, and a warning is emitted. The attribute is also overridden by a subsequent declaration as `dllexport`.

When applied to C++ classes, the attribute marks non-inlined member functions and static data members as imports. However, the attribute is ignored for virtual methods to allow creation of vtables using thunks.

On cygwin, mingw and arm-pe targets, `__declspec(dllimport)` is recognized as a synonym for `__attribute__ ((dllimport))` for compatibility with other Microsoft Windows compilers.

The use of the `dllimport` attribute on functions is not necessary, but provides a small performance benefit by eliminating a thunk in the dll. The use of the `dllimport` attribute on imported variables was required on older versions of GNU ld, but can now be avoided by passing the `-enable-auto-import` switch to ld. As with functions, using the attribute for a variable eliminates a thunk in the dll.

One drawback to using this attribute is that a pointer to a function or variable marked as dllimport cannot be used as a constant address. The attribute can be disabled for functions by setting the `-mnop-fun-dllimport` flag.

`dllexport`

On Microsoft Windows targets the `dllexport` attribute causes the compiler to provide a global pointer to a pointer in a dll, so that it can be referenced with the `dllimport` attribute. The pointer name is formed by combining `_imp__` and the function or variable name.

Currently, the `dllexport`attribute is ignored for inlined functions, but export can be forced by using the `-fkeep-inline-functions` flag. The attribute is also ignored for undefined symbols.

When applied to C++ classes. the attribute marks defined non-inlined member functions and static data members as exports. Static consts initialized in-class are not marked unless they are also defined out-of-class.

On cygwin, mingw and arm-pe targets, `__declspec(dllexport)` is recognized as a synonym for `__attribute__ ((dllexport))` for compatibility with other Microsoft Windows compilers.

Alternative methods for including the symbol in the dll's export table are to use a .def file with an `EXPORTS` section or, with GNU ld, using the `-export-all` linker flag.

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.

6.26. 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.

Section 6.25 *Declaring Attributes of Functions*, for details of the semantics of attributes applying to functions. Section 6.32 *Specifying Attributes of Variables*, for details of the semantics of attributes applying to variables. Section 6.33 *Specifying Attributes of Types*, 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.

In GNU C, 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. GNU C++ does not permit such placement of attribute lists, as it is permissible for a declaration, which could begin with an attribute list, to be labelled in C++. Declarations cannot be labelled in C90 or C99, so the ambiguity does not arise there.

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 (Section 6.37 *Controlling Names Used in Assembler Code*), 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.

6.27. 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.

6.28. 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 included in the 1999 C standard. However, C++ style comments are not recognized if you specify an `-std` option specifying a version of ISO C before C99, or `-ansi` (equivalent to `-std=c89`).

6.29. 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.

6.30. The Character [ESC] in Constants

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

6.31. 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 address. 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 (Section 6.32 *Specifying Attributes of Variables*). 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.

6.32. 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. Some attributes are currently defined generically for variables. Other attributes are defined for variables on particular target systems. Other attributes are available for functions (Section 6.25 *Declaring Attributes of Functions*) and for types (Section 6.33 *Specifying Attributes of Types*). Other front ends might define more attributes (Chapter 7 *Extensions to the C++ Language*).

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`.

Section 6.26 *Attribute Syntax*, 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.

`cleanup (cleanup_function)`

The `cleanup` attribute runs a function when the variable goes out of scope. This attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.

If `-fexceptions` is enabled, then `cleanup_function` will be run during the stack unwinding that happens during the processing of the exception. Note that the `cleanup` attribute does not allow the exception to be caught, only to perform an action. It is undefined what happens if `cleanup_function` does not return normally.

`common`
`nocommon`

The `common` attribute requests GCC to place a variable in "common" storage. The `nocommon` attribute requests the opposite - to allocate space for it directly.

These attributes override the default chosen by the `-fno-common` and `-fcommon` flags respectively.

`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 warning 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 (Section 6.25 *Declaring Attributes of Functions*, Section 6.33 *Specifying Attributes of Types*.)

`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.

`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 Microsoft Windows, 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 Microsoft Windows.

`tls_model ("tls_model")`

The `tls_model` attribute sets thread-local storage model (Section 6.49 *Thread-Local Storage*) of a particular `__thread` variable, overriding `-ftls-model=` command line switch on a per-variable basis. The `tls_model` argument should be one of `global-dynamic`, `local-dynamic`, `initial-exec` or `local-exec`.

Not all targets support this attribute.

`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 Section 6.33 *Specifying Attributes of Types*. 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.

`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 Section 6.25 *Declaring Attributes of Functions*.

`dllimport`

The `dllimport` attribute is described in Section 6.25 *Declaring Attributes of Functions*.

`dlexport`

The `dllexport` attribute is described in Section 6.25 *Declaring Attributes of Functions*.

6.32.1. M32R/D Variable Attributes

One attribute is currently defined for the M32R/D.

`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).

6.32.2. i386 Variable Attributes

Two attributes are currently defined for i386 configurations: `ms_struct` and `gcc_struct`

`ms_struct`
`gcc_struct`

If `packed` is used on a structure, or if bit-fields are used it may be that the Microsoft ABI packs them differently than GCC would normally pack them. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

Currently `-m[no-]ms-bitfields` is provided for the Microsoft Windows X86 compilers to match the native Microsoft compiler.

6.33. 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. Six attributes are currently defined for types: `aligned`, `packed`, `transparent_union`, `unused`, `deprecated` and `may_alias`. Other attributes are defined for functions (Section 6.25 *Declaring Attributes of Functions*) and for variables (Section 6.32 *Specifying Attributes of Variables*).

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

You may specify the `aligned` and `transparent_union` attributes either in a `typedef` declaration or just past the closing curly brace of a complete enum, struct or union type *definition* and the `packed` attribute only past the closing brace of a definition.

You may also specify attributes between the enum, struct or union tag and the name of the type rather than after the closing brace.

Section 6.26 *Attribute Syntax*, 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 `struct` or `union` type definition, specifies that each member of the structure or union is placed to minimize the memory required. When attached to an `enum` definition, it indicates that the smallest integral type should be used.

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.

In the following example `struct my_packed_struct`'s members are packed closely together, but the internal layout of its `s` member is not packed - to do that, `struct my_unpacked_struct` would need to be packed too.

```
struct my_unpacked_struct
 {
    char c;
    int i;
 };

struct my_packed_struct __attribute__ ((__packed__))
  {
     char c;
     int  i;
     struct my_unpacked_struct s;
  };
```

You may only specify this attribute on the definition of a `enum`, `struct` or `union`, not on a `typedef` which does not also define the enumerated type, structure or union.

`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 the 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 (Section 6.25 *Declaring Attributes of Functions*, Section 6.32 *Specifying Attributes of Variables*.)

`may_alias`

Accesses to objects with types with this attribute are not subjected to type-based alias analysis, but are instead assumed to be able to alias any other type of objects, just like the `char` type. See `-fstrict-aliasing` for more information on aliasing issues.

Example of use:

```
typedef short __attribute__((__may_alias__)) short_a;

int
main (void)
{
  int a = 0x12345678;
  short_a *b = (short_a *) &a;

  b[1] = 0;

  if (a == 0x12345678)
    abort();

  exit(0);
}
```

If you replaced `short_a` with `short` in the variable declaration, the above program would abort when compiled with `-fstrict-aliasing`, which is on by default at `-O2` or above in recent GCC versions.

6.33.1. i386 Type Attributes

Two attributes are currently defined for i386 configurations: `ms_struct` and `gcc_struct`

`ms_struct`
`gcc_struct`

If `packed` is used on a structure, or if bit-fields are used it may be that the Microsoft ABI packs them differently than GCC would normally pack them. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

Currently `-m[no-]ms-bitfields` is provided for the Microsoft Windows X86 compilers to match the native Microsoft compiler.

To specify multiple attributes, separate them by commas within the double parentheses: for example, `__attribute__ ((aligned (16), packed))`.

6.34. 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`. Section 6.39 *Alternate Keywords*.) You can also make all "simple enough" functions inline with the option `-finline-functions`.

Note that certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of varargs, use of alloca, use of variable sized data types (Section 6.14 *Arrays of Variable Length*), use of computed goto (Section 6.3 *Labels as Values*), use of nonlocal goto, and nested functions (Section 6.4 *Nested Functions*). 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`; Section 4.5 *Options Controlling C++ Dialect*.)

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, 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.

Since GCC eventually will implement ISO C99 semantics for inline functions, it is best to use `static inline` only to guarantee compatibility. (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));
```

6.35. 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 (Section 6.36 *Constraints for `asm`Operands*).

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 you 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. You should only use read-write operands when the constraints for the operand (or the operand in which only some of the bits are to be changed) allow 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. Variables declared to live in specific registers (Section 6.38 *Variables in Specified Registers*), and used as asm input or output operands must have no part mentioned in the clobber description. 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 instructions access 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 and not optimize stores or loads to that memory. 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`. If you know how large the accessed memory is, you can add it as input or output but if this is not known, you should add `memory`. As an example, if you access ten bytes of a string, you can use a memory input like:

```
{"m"( ({ struct { char x[10]; } *p = (void *)ptr ; *p; }) )}.
```

Note that in the following example the memory input is necessary, otherwise GCC might optimize the store to `x` away:

```
int foo ()
{
  int x = 42;
  int *y = &x;
  int result;
  asm ("magic stuff accessing an 'int' pointed to by '%1'"
        "=&d" (r) : "a" (y), "m" (*y));
  return result;
}
```

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. Section 6.36.3 *Constraint Modifier Characters*.

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`. Section 6.39 *Alternate Keywords*.

6.35.1. Size of an `asm`

Some targets require that GCC track the size of each instruction used in order to generate correct code. Because the final length of an `asm` is only known by the assembler, GCC must make an estimate as to how big it will be. The estimate is formed by counting the number of statements in the pattern of the `asm` and multiplying that by the length of the longest instruction on that processor. Statements in the `asm` are identified by newline characters and whatever statement separator characters are supported by the assembler; on most processors this is the '`;`' character.

Normally, GCC's estimate is perfectly adequate to ensure that correct code is generated, but it is possible to confuse the compiler if you use pseudo instructions or assembler macros that expand into multiple real instructions or if you use assembler directives that expand to more space in the object file than would be needed for a single instruction. If this happens then the assembler will produce a diagnostic saying that a label is unreachable.

6.35.2. 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)");
```

6.36. 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.

6.36.1. 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` or `const_vector`) 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 consecutively, 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.

6.36.2. 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.

6.36.3. 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. GCC can only handle one commutative pair in an asm; if you use more, the compiler may fail. Note that you need not use the modifier if the two alternatives are strictly identical; this would only waste time in the reload pass.

`#`

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.

6.36.4. 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; Section 6.36.1 *Simple Constraints*), 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 lowercase).

`CONST_OK_FOR_LETTER_P`

Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually uppercase).

`CONST_DOUBLE_OK_FOR_LETTER_P`

Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually uppercase).

`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

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

PowerPC and IBM RS6000--`rs6000.h`

`b`
: Address base register

`f`
: Floating point register

`v`
: Vector 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

`C`

Specifies constant that can be easily constructed in SSE register without loading it from memory.

`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

Intel IA-64--`ia64.h`

`a`
: General register `r0` to `r3` for `addl` instruction

`b`
: Branch register

`c`
: Predicate register (`c` as in "conditional")

`d`
: Application register residing in M-unit

`e`
: Application register residing in I-unit

`f`
: Floating-point register

`m`
: Memory operand. Remember that `m` allows postincrement and postdecrement which require printing with `%Pn` on IA-64. Use `S` to disallow postincrement and postdecrement.

`G`
: Floating-point constant 0.0 or 1.0

`I`
: 14-bit signed integer constant

`J`

22-bit signed integer constant

`K`

8-bit signed integer constant for logical instructions

`L`

8-bit adjusted signed integer constant for compare pseudo-ops

`M`

6-bit unsigned integer constant for shift counts

`N`

9-bit signed integer constant for load and store postincrements

`O`

The constant zero

`P`

0 or -1 for `dep` instruction

`Q`

Non-volatile memory for floating-point loads and stores

`R`

Integer constant in the range 1 to 4 for `shladd` instruction

`S`

Memory operand except postincrement and postdecrement

IP2K--`ip2k.h`

`a`

`DP` or `IP` registers (general address)

`f`

`IP` register

`j`

`IPL` register

`k`

`IPH` register

`b`

`DP` register

`y`
: `DPH` register

`z`
: `DPL` register

`q`
: `SP` register

`c`
: `DP` or `SP` registers (offsettable address)

`d`
: Non-pointer registers (not `SP`, `DP`, `IP`)

`u`
: Non-SP registers (everything except `SP`)

`R`
: Indirect through `IP` - Avoid this except for `QImode`, since we can't access extra bytes

`S`
: Indirect through `SP` or `DP` with short displacement (0..127)

`T`
: Data-section immediate value

`I`
: Integers from -255 to -1

`J`
: Integers from 0 to 7--valid bit number in a register

`K`
: Integers from 0 to 127--valid displacement for addressing mode

`L`
: Integers from 1 to 127

`M`
: Integer -1

`N`
: Integer 1

`O`
: Zero

`P`

Integers from 0 to 255

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

`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

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 on the SPARC-V8 architecture and lower floating-point register on the SPARC-V9 architecture.

`e`

Floating-point register. It is equivalent to `f` on the SPARC-V8 architecture and contains both lower and upper floating-point registers on the SPARC-V9 architecture.

`c`

Floating-point condition code register.

`d`

Lower floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.

`b`

Floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.

`h`

64-bit global or out register for the SPARC-V8+ architecture.

`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`

`O`

The constant 4096

`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`

Value appropriate as displacement.

`(0..4095)`

for short displacement

`(-524288..524287)`

for long displacement

`M`

Constant integer with a value of 0x7fffffff.

`N`

Multiple letter constraint followed by 4 parameter letters.

`0..9:`

number of the part counting from most to least significant

`H,Q:`

mode of the part

`D,S,H:`

mode of the containing operand

`0,F:`

value of the other parts (F - all bits set)

The constraint matches if the specified part of a constant has a value different from it's other parts.

`Q`

Memory reference without index register and with short displacement.

`R`

Memory reference with index register and short displacement.

`S`

Memory reference without index register but with long displacement.

`T`

Memory reference with index register and long displacement.

`U`

Pointer with short displacement.

`W`

Pointer with long displacement.

`Y`

Shift count operand.

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 a 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.

`Z`

The constant 0.

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

6.37. 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 6.38 *Variables in Specified Registers*. 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.

6.38. 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 (Section 6.35 *Assembler Instructions with C Expression Operands*), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

6.38.1. 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.

6.38.2. 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 (Section 6.35 *Assembler Instructions with C Expression Operands*). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. However, these registers are made unavailable for use in the reload pass; excessive use of this feature 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.

6.39. Alternate Keywords

`-ansi` and the various `-std` options disable certain keywords. This causes trouble when you want to use GNU C extensions, or a general-purpose header file that should be usable by all programs, including ISO C programs. The keywords `asm`, `typeof` and `inline` are not available in programs compiled with `-ansi` or `-std` (although `inline` can be used in a program compiled with `-std=c99`). 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`, 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.

6.40. 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++.

6.41. Function Names as Strings

GCC provides three magic variables which hold the name of the current function, as a string. The first of these is `__func__`, which is part of the C99 standard:

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.

`__FUNCTION__` is another name for `__func__`. Older versions of GCC recognize only this name. However, it is not standardized. For maximum portability, we recommend you use `__func__`, but provide a fallback definition with the preprocessor:

```
#if __STDC_VERSION__ < 199901L
# if __GNUC__ >= 2
#  define __func__ __FUNCTION__
# else
#  define __func__ "<unknown>"
# endif
#endif
```

In C, `__PRETTY_FUNCTION__` is yet another name for `__func__`. However, in C++, `__PRETTY_FUNCTION__` contains the type signature of the function as well as its bare name. For example, this program:

```
extern "C" {
extern int printf (char *, ...);
}

class a {
 public:
  void 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__ = void a::sub(int)
```

These identifiers are not preprocessor macros. In GCC 3.3 and earlier, in C only, `__FUNCTION__` and `__PRETTY_FUNCTION__` were treated as string literals; they could be used to initialize `char` arrays, and they could be concatenated with other string literals. GCC 3.4 and later treat them as variables, like `__func__`. In C++, `__FUNCTION__` and `__PRETTY_FUNCTION__` have always been variables.

6.42. 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`) 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. When inlining the expected behavior is that the function will return the address of the function that will be returned to. To work around this behavior use the `noinline` function attribute.

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`) 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.

6.43. 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 `VnB`; `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.

Specifying a combination that is not valid for the current architecture will cause GCC to synthesize the instructions using a narrower mode. For example, if you specify a variable of type `V4SI` and your architecture does not allow for this specific SIMD type, GCC will produce code that uses 4 `SIs`.

The types defined in this manner can be used with a subset of normal C operations. Currently, GCC will allow using the following operators on these types: `+, -, *, /, unary minus, ^, |, &, ~`.

The operations behave like C++ `valarrays`. Addition is defined as the addition of the corresponding elements of the operands. For example, in the code below, each of the 4 elements in `a` will be added to the corresponding 4 elements in `b` and the resulting vector will be stored in `c`.

```
typedef int v4si __attribute__ ((mode(V4SI)));

v4si a, b, c;

c = a + b;
```

Subtraction, multiplication, division, and the logical operations operate in a similar manner. Likewise, the result of using the unary minus or complement operators on a vector type is a vector whose elements are the negative or complemented values of the corresponding elements in the operand.

You can declare variables and use them in function calls and returns, as well as in assignments and some casts. You can specify a vector type as a return type for a function. Vector types can also be used as function arguments. 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).

You cannot operate between vectors of different lengths or different signedness without a cast.

A port that supports hardware vector operations, usually 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);
}
```

6.44. Object Size Checking Builtins

GCC implements a limited buffer overflow protection mechanism that can prevent some buffer overflow attacks.

size_t `__builtin_object_size` (void * `ptr`, int `type`) is a built-in construct that returns a constant number of bytes from `ptr` to the end of the object `ptr` pointer points to (if known at compile time). `__builtin_object_size` never evaluates its arguments for side-effects. If there are any side-effects in them, it returns `(size_t) -1` for `type` 0 or 1 and `(size_t) 0` for `type` 2 or 3. If there are multiple objects `ptr` can point to and all of them are known at compile time, the returned number is the maximum of remaining byte counts in those objects if `type` & 2 is 0 and minimum if non-zero. If it is not possible to determine which objects `ptr` points to at compile time, `__builtin_object_size` should return `(size_t) -1` for `type` 0 or 1 and `(size_t) 0` for `type` 2 or 3.

`type` is an integer constant from 0 to 3. If the least significant bit is clear, objects are whole variables, if it is set, a closest surrounding subobject is considered the object a pointer points to. The second bit determines if maximum or minimum of remaining bytes is computed.

```
struct V { char buf1[10]; int b; char buf2[10]; } var;
char *p = &var.buf1[1], *q = &var.b;

/* Here the object p points to is var.  */
assert (__builtin_object_size (p, 0) == sizeof (var) - 1);
/* The subobject p points to is var.buf1.  */
assert (__builtin_object_size (p, 1) == sizeof (var.buf1) - 1);
/* The object q points to is var.  */
assert (__builtin_object_size (q, 0)
 == (char *) (&var + 1) - (char *) &var.b);
/* The subobject q points to is var.b.  */
assert (__builtin_object_size (q, 1) == sizeof (var.b));
```

There are built-in functions added for many common string operation functions, e.g. for `memcpy` `__builtin___memcpy_chk` built-in is provided. This built-in has an additional last argument, which is the number of bytes remaining in object the `dest` argument points to or `(size_t) -1` if the size is not known.

The built-in functions are optimized into the normal string functions like `memcpy` if the last argument is `(size_t) -1` or if it is known at compile time that the destination object will not be overflown. If the compiler can determine at compile time the object will be always overflown, it issues a warning.

The intended use can be e.g.

```
#undef memcpy
#define bos0(dest) __builtin_object_size (dest, 0)
#define memcpy(dest, src, n) \
  __builtin___memcpy_chk (dest, src, n, bos0 (dest))

char *volatile p;
char buf[10];
/* It is unknown what object p points to, so this is optimized
   into plain memcpy - no checking is possible.  */
memcpy (p, "abcde", n);
/* Destination is known and length too.  It is known at compile
   time there will be no overflow.  */
memcpy (&buf[5], "abcde", 5);
/* Destination is known, but the length is not known at compile time.
   This will result in __memcpy_chk call that can check for overflow
   at runtime.  */
memcpy (&buf[5], "abcde", n);
/* Destination is known and it is known at compile time there will
   be overflow.  There will be a warning and __memcpy_chk call that
   will abort the program at runtime.  */
memcpy (&buf[6], "abcde", 5);
```

Such built-in functions are provided for `memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat` and `strncat`.

There are also checking built-in functions for formatted output functions.

```
int __builtin___sprintf_chk (char *s, int flag, size_t os, const char *fmt, ...);
int __builtin___snprintf_chk (char *s, size_t maxlen, int flag, size_t os,
         const char *fmt, ...);
int __builtin___vsprintf_chk (char *s, int flag, size_t os, const char *fmt,
         va_list ap);
int __builtin___vsnprintf_chk (char *s, size_t maxlen, int flag, size_t os,
          const char *fmt, va_list ap);
```

The added `flag` argument is passed unchanged to `__sprintf_chk` etc. functions and can contain implementation specific flags on what additional security measures the checking function might take, such as handling `%n` differently.

The `os` argument is the object size `s` points to, like in the other built-in functions. There is a small difference in the behaviour though, if `os` is `(size_t) -1`, the built-in functions are optimized into the non-checking functions only if `flag` is 0, otherwise the checking function is called with `os` argument set to `(size_t) -1`.

In addition to this, there are checking built-in functions `__builtin___printf_chk`, `__builtin___vprintf_chk`, `__builtin___fprintf_chk` and `__builtin___vfprintf_chk`. These have just one additional argument, `flag`, right before format string `fmt`. If the compiler is able to optimize them to `fputc` etc. functions, it will, otherwise the checking function should be called and the `flag` argument passed to it.

6.45. 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. (Section 4.4 *Options Controlling C Dialect*) 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.

Outside strict ISO C mode (`-ansi`, `-std=c89` or `-std=c99`), the functions `_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `dremf`, `dreml`, `drem`, `exp10f`, `exp10l`, `exp10`, `ffsll`, `ffsl`, `ffs`, `fprintf_unlocked`, `fputs_unlocked`, `gammaf`, `gammal`, `gamma`, `gettext`, `index`, `j0f`, `j0l`, `j0`, `j1f`, `j1l`, `j1`, `jnf`, `jnl`, `jn`, `mempcpy`, `pow10f`, `pow10l`, `pow10`, `printf_unlocked`, `rindex`, `scalbf`, `scalbl`, `scalb`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `strdup`, `strfmon`, `y0f`, `y0l`, `y0`, `y1f`, `y1l`, `y1`, `ynf`, `ynl` and `yn` 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 `_Exit`, `acoshf`, `acoshl`, `acosh`, `asinhf`, `asinhl`, `asinh`, `atanhf`, `atanhl`, `atanh`, `cabsf`, `cabsl`, `cabs`, `cacosf`, `cacoshf`, `cacoshl`, `cacosh`, `cacosl`, `cacos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhf`, `casinhl`, `casinh`, `casinl`, `casin`, `catanf`, `catanhf`, `catanhl`, `catanh`, `catanl`, `catan`, `cbrtf`, `cbrtl`, `cbrt`, `ccosf`, `ccoshf`, `ccoshl`, `ccosh`, `ccosl`, `ccos`, `cexpf`, `cexpl`, `cexp`, `cimagf`, `cimagl`, `cimag`, `conjf`, `conjl`, `conj`, `copysignf`, `copysignl`, `copysign`, `cpowf`, `cpowl`, `cpow`, `cprojf`, `cprojl`, `cproj`, `crealf`, `creall`, `creal`, `csinf`, `csinhf`, `csinhl`, `csinh`, `csinl`, `csin`, `csqrtf`, `csqrtl`, `csqrt`, `ctanf`, `ctanhf`, `ctanhl`, `ctanh`, `ctanl`, `ctan`, `erfcf`, `erfcl`, `erfc`, `erff`, `erfl`, `erf`, `exp2f`, `exp2l`, `exp2`, `expm1f`, `expm1l`, `expm1`, `fdimf`, `fdiml`, `fdim`, `fmaf`, `fmal`, `fmaxf`, `fmaxl`, `fmax`, `fma`, `fminf`, `fminl`, `fmin`, `hypotf`, `hypotl`, `hypot`, `ilogbf`, `ilogbl`, `ilogb`, `imaxabs`, `lgammaf`, `lgammal`, `lgamma`, `llabs`, `llrintf`, `llrintl`, `llrint`, `llroundf`, `llroundl`, `llround`, `log1pf`, `log1pl`, `log1p`, `log2f`, `log2l`, `log2`, `logbf`, `logbl`, `logb`, `lrintf`, `lrintl`, `lrint`, `lroundf`, `lroundl`, `lround`, `nearbyintf`, `nearbyintl`, `nearbyint`, `nextafterf`, `nextafterl`, `nextafter`, `nexttowardf`, `nexttowardl`, `nexttoward`, `remainderf`, `remainderl`, `remainder`, `remquof`, `remquol`, `remquo`, `rintf`, `rintl`, `rint`, `roundf`, `roundl`, `round`, `scalblnf`, `scalblnl`, `scalbln`, `scalbnf`, `scalbnl`, `scalbn`, `snprintf`, `tgammaf`, `tgammal`, `tgamma`, `truncf`, `truncl`, `trunc`, `vfscanf`, `vscanf`, `vsnprintf` and `vsscanf` are handled as built-in functions except in strict ISO C90 mode (`-ansi` or `-std=c89`).

There are also built-in versions of the ISO C99 functions `acosf`, `acosl`, `asinf`, `asinl`, `atan2f`, `atan2l`, `atanf`, `atanl`, `ceilf`, `ceill`, `cosf`, `coshf`, `coshl`, `cosl`, `expf`, `expl`, `fabsf`, `fabsl`, `floorf`, `floorl`, `fmodf`, `fmodl`, `frexpf`, `frexpl`, `ldexpf`, `ldexpl`, `log10f`, `log10l`, `logf`, `logl`, `modfl`, `modf`, `powf`, `powl`, `sinf`, `sinhf`, `sinhl`, `sinl`, `sqrtf`, `sqrtl`, `tanf`, `tanhf`, `tanhl` and `tanl` that are recognized in any mode since ISO C90 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with `__builtin_`.

The ISO C90 functions `abort`, `abs`, `acos`, `asin`, `atan2`, `atan`, `calloc`, `ceil`, `cosh`, `cos`, `exit`, `exp`, `fabs`, `floor`, `fmod`, `fprintf`, `fputs`, `frexp`, `fscanf`, `labs`, `ldexp`, `log10`, `log`, `malloc`, `memcmp`, `memcpy`, `memset`, `modf`, `pow`, `printf`, `putchar`, `puts`, `scanf`, `sinh`, `sin`, `snprintf`, `sprintf`, `sqrt`, `sscanf`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `tanh`, `tan`, `vfprintf`, `vprintf` and `vsprintf` 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`) 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 not considered to be compatible with another `enum` type even if both are compatible with the same integer type; this is what the C standard specifies. For example, `enum {foo, bar}` is not 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`) 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) 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 (Section 6.20 *Compound Literals*) 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`) 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`, ...) 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.

double`__builtin_huge_val` (void) Returns a positive infinity, if supported by the floating-point format, else `DBL_MAX`. This function is suitable for implementing the ISO C macro `HUGE_VAL`.

float`__builtin_huge_valf` (void) Similar to `__builtin_huge_val`, except the return type is `float`.

long double`__builtin_huge_vall` (void) Similar to `__builtin_huge_val`, except the return type is `long double`.

double__builtin_inf (void) Similar to __builtin_huge_val, except a warning is generated if the target floating-point format does not support infinities. This function is suitable for implementing the ISO C99 macro INFINITY.

float__builtin_inff (void) Similar to __builtin_inf, except the return type is float.

long double__builtin_infl (void) Similar to __builtin_inf, except the return type is long double.

double__builtin_nan (const char *str) This is an implementation of the ISO C99 function nan.

Since ISO C99 defines this function in terms of strtod, which we do not implement, a description of the parsing is in order. The string is parsed as by strtol; that is, the base is recognized by leading 0 or 0x prefixes. The number parsed is placed in the significand such that the least significant bit of the number is at the least significant bit of the significand. The number is truncated to fit the significand field provided. The significand is forced to be a quiet NaN.

This function, if given a string literal, is evaluated early enough that it is considered a compile-time constant.

float__builtin_nanf (const char *str) Similar to __builtin_nan, except the return type is float.

long double__builtin_nanl (const char *str) Similar to __builtin_nan, except the return type is long double.

double__builtin_nans (const char *str) Similar to __builtin_nan, except the significand is forced to be a signaling NaN. The nans function is proposed by http://www.open-std.org/jtc1/sc22/wg14/www/docs/n965.htmWG14 N965.

float__builtin_nansf (const char *str) Similar to __builtin_nans, except the return type is float.

long double__builtin_nansl (const char *str) Similar to __builtin_nans, except the return type is long double.

int__builtin_ffs (unsigned int x) Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

int__builtin_clz (unsigned int x) Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

int__builtin_ctz (unsigned int x) Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

int__builtin_popcount (unsigned int x) Returns the number of 1-bits in x.

int__builtin_parity (unsigned int x) Returns the parity of x, i.e. the number of 1-bits in x modulo 2.

int__builtin_ffsl (unsigned long) Similar to __builtin_ffs, except the argument type is unsigned long.

int__builtin_clzl (unsigned long) Similar to __builtin_clz, except the argument type is unsigned long.

int__builtin_ctzl (unsigned long) Similar to __builtin_ctz, except the argument type is unsigned long.

int__builtin_popcountl (unsigned long) Similar to __builtin_popcount, except the argument type is unsigned long.

int__builtin_parityl (unsigned long) Similar to __builtin_parity, except the argument type is unsigned long.

int__builtin_ffsll (unsigned long long) Similar to __builtin_ffs, except the argument type is unsigned long long.

int__builtin_clzll (unsigned long long) Similar to __builtin_clz, except the argument type is unsigned long long.

int__builtin_ctzll (unsigned long long) Similar to __builtin_ctz, except the argument type is unsigned long long.

int__builtin_popcountll (unsigned long long) Similar to __builtin_popcount, except the argument type is unsigned long long.

int__builtin_parityll (unsigned long long) Similar to __builtin_parity, except the argument type is unsigned long long.

6.46. 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.

6.46.1. 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 (Section 6.43 *Using vector instructions through built-in functions*): 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_cmpunordss (v4sf, v4sf)
v4si __builtin_ia32_cmpneqss (v4sf, v4sf)
v4si __builtin_ia32_cmpnlts (v4sf, v4sf)
v4si __builtin_ia32_cmpnless (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 `-msse3` is used. All of them generate the machine instruction that is part of the name.

```
v2df __builtin_ia32_addsubpd (v2df, v2df)
v2df __builtin_ia32_addsubps (v2df, v2df)
v2df __builtin_ia32_haddpd (v2df, v2df)
v2df __builtin_ia32_haddps (v2df, v2df)
v2df __builtin_ia32_hsubpd (v2df, v2df)
v2df __builtin_ia32_hsubps (v2df, v2df)
v16qi __builtin_ia32_lddqu (char const *)
void __builtin_ia32_monitor (void *, unsigned int, unsigned int)
v2df __builtin_ia32_movddup (v2df)
v4sf __builtin_ia32_movshdup (v4sf)
v4sf __builtin_ia32_movsldup (v4sf)
void __builtin_ia32_mwait (unsigned int, unsigned int)
```

The following built-in functions are available when `-msse3` is used.

`v2df __builtin_ia32_loadddup (double const *)`

Generates the `movddup` machine instruction as a load from 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)
```

6.46.2. PowerPC AltiVec Built-in Functions

GCC provides an interface for the PowerPC family of processors to access the AltiVec operations described in Motorola's AltiVec Programming Interface Manual. The interface is made available by including `<altivec.h>` and using `-maltivec` and `-mabi=altivec`. The interface supports the following vector types.

```
vector unsigned char
vector signed char
vector bool char

vector unsigned short
vector signed short
vector bool short
vector pixel

vector unsigned int
vector signed int
vector bool int
vector float
```

GCC's implementation of the high-level language interface available from C and C++ code differs from Motorola's documentation in several ways.

- A vector constant is a list of constant expressions within curly braces.
- A vector initializer requires no cast if the vector constant is of the same type as the variable it is initializing.

- If `signed` or `unsigned` is omitted, the vector type defaults to `signed` for `vector int` or `vector short` and to `unsigned` for `vector char`.
- Compiling with `-maltivec` adds keywords `__vector`, `__pixel`, and `__bool`. Macros `vector`, `pixel`, and `bool` are defined in `<altivec.h>` and can be undefined.
- GCC allows using a `typedef` name as the type specifier for a vector type.
- For C, overloaded functions are implemented with macros so the following does not work:

  ```
  vec_add ((vector signed int){1, 2, 3, 4}, foo);
  ```

  Since `vec_add` is a macro, the vector constant in the example is treated as four separate arguments. Wrap the entire argument in parentheses for this to work.

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.

The following interfaces are supported for the generic and specific AltiVec operations and the AltiVec predicates. In cases where there is a direct mapping between generic and specific operations, only the generic names are shown here, although the specific operations can also be used.

Arguments that are documented as `const int` require literal integral values within the range required for that operation.

```
vector signed char vec_abs (vector signed char);
vector signed short vec_abs (vector signed short);
vector signed int vec_abs (vector signed int);
vector float vec_abs (vector float);

vector signed char vec_abss (vector signed char);
vector signed short vec_abss (vector signed short);
vector signed int vec_abss (vector signed int);

vector signed char vec_add (vector bool char, vector signed char);
vector signed char vec_add (vector signed char, vector bool char);
vector signed char vec_add (vector signed char, vector signed char);
vector unsigned char vec_add (vector bool char, vector unsigned char);
vector unsigned char vec_add (vector unsigned char, vector bool char);
vector unsigned char vec_add (vector unsigned char,
                              vector unsigned char);
vector signed short vec_add (vector bool short, vector signed short);
vector signed short vec_add (vector signed short, vector bool short);
vector signed short vec_add (vector signed short, vector signed short);
vector unsigned short vec_add (vector bool short,
                               vector unsigned short);
vector unsigned short vec_add (vector unsigned short,
                               vector bool short);
vector unsigned short vec_add (vector unsigned short,
                               vector unsigned short);
vector signed int vec_add (vector bool int, vector signed int);
vector signed int vec_add (vector signed int, vector bool int);
vector signed int vec_add (vector signed int, vector signed int);
vector unsigned int vec_add (vector bool int, vector unsigned int);
vector unsigned int vec_add (vector unsigned int, vector bool int);
vector unsigned int vec_add (vector unsigned int, vector unsigned int);
vector float vec_add (vector float, vector float);

vector float vec_vaddfp (vector float, vector float);

vector signed int vec_vadduwm (vector bool int, vector signed int);
vector signed int vec_vadduwm (vector signed int, vector bool int);
vector signed int vec_vadduwm (vector signed int, vector signed int);
```

```
vector unsigned int vec_vadduwm (vector bool int, vector unsigned int);
vector unsigned int vec_vadduwm (vector unsigned int, vector bool int);
vector unsigned int vec_vadduwm (vector unsigned int,
                                 vector unsigned int);

vector signed short vec_vadduhm (vector bool short,
                                 vector signed short);
vector signed short vec_vadduhm (vector signed short,
                                 vector bool short);
vector signed short vec_vadduhm (vector signed short,
                                 vector signed short);
vector unsigned short vec_vadduhm (vector bool short,
                                   vector unsigned short);
vector unsigned short vec_vadduhm (vector unsigned short,
                                   vector bool short);
vector unsigned short vec_vadduhm (vector unsigned short,
                                   vector unsigned short);

vector signed char vec_vaddubm (vector bool char, vector signed char);
vector signed char vec_vaddubm (vector signed char, vector bool char);
vector signed char vec_vaddubm (vector signed char, vector signed char);
vector unsigned char vec_vaddubm (vector bool char,
                                  vector unsigned char);
vector unsigned char vec_vaddubm (vector unsigned char,
                                  vector bool char);
vector unsigned char vec_vaddubm (vector unsigned char,
                                  vector unsigned char);

vector unsigned int vec_addc (vector unsigned int, vector unsigned int);

vector unsigned char vec_adds (vector bool char, vector unsigned char);
vector unsigned char vec_adds (vector unsigned char, vector bool char);
vector unsigned char vec_adds (vector unsigned char,
                               vector unsigned char);
vector signed char vec_adds (vector bool char, vector signed char);
vector signed char vec_adds (vector signed char, vector bool char);
vector signed char vec_adds (vector signed char, vector signed char);
vector unsigned short vec_adds (vector bool short,
                                vector unsigned short);
vector unsigned short vec_adds (vector unsigned short,
                                vector bool short);
vector unsigned short vec_adds (vector unsigned short,
                                vector unsigned short);
vector signed short vec_adds (vector bool short, vector signed short);
vector signed short vec_adds (vector signed short, vector bool short);
vector signed short vec_adds (vector signed short, vector signed short);
vector unsigned int vec_adds (vector bool int, vector unsigned int);
vector unsigned int vec_adds (vector unsigned int, vector bool int);
vector unsigned int vec_adds (vector unsigned int, vector unsigned int);
vector signed int vec_adds (vector bool int, vector signed int);
vector signed int vec_adds (vector signed int, vector bool int);
vector signed int vec_adds (vector signed int, vector signed int);

vector signed int vec_vaddsws (vector bool int, vector signed int);
vector signed int vec_vaddsws (vector signed int, vector bool int);
vector signed int vec_vaddsws (vector signed int, vector signed int);

vector unsigned int vec_vadduws (vector bool int, vector unsigned int);
vector unsigned int vec_vadduws (vector unsigned int, vector bool int);
vector unsigned int vec_vadduws (vector unsigned int,
                                 vector unsigned int);
vector signed short vec_vaddshs (vector bool short,
                                 vector signed short);
vector signed short vec_vaddshs (vector signed short,
                                 vector bool short);
vector signed short vec_vaddshs (vector signed short,
                                 vector signed short);

vector unsigned short vec_vadduhs (vector bool short,
                                   vector unsigned short);
vector unsigned short vec_vadduhs (vector unsigned short,
                                   vector bool short);
vector unsigned short vec_vadduhs (vector unsigned short,
                                   vector unsigned short);

vector signed char vec_vaddsbs (vector bool char, vector signed char);
vector signed char vec_vaddsbs (vector signed char, vector bool char);
vector signed char vec_vaddsbs (vector signed char, vector signed char);

vector unsigned char vec_vaddubs (vector bool char,
                                  vector unsigned char);
vector unsigned char vec_vaddubs (vector unsigned char,
                                  vector bool char);
vector unsigned char vec_vaddubs (vector unsigned char,
                                  vector unsigned char);

vector float vec_and (vector float, vector float);
vector float vec_and (vector float, vector bool int);
vector float vec_and (vector bool int, vector float);
vector bool int vec_and (vector bool int, vector bool int);
vector signed int vec_and (vector bool int, vector signed int);
vector signed int vec_and (vector signed int, vector bool int);
vector signed int vec_and (vector signed int, vector signed int);
vector unsigned int vec_and (vector bool int, vector unsigned int);
vector unsigned int vec_and (vector unsigned int, vector bool int);
vector unsigned int vec_and (vector unsigned int, vector unsigned int);
vector bool short vec_and (vector bool short, vector bool short);
vector signed short vec_and (vector bool short, vector signed short);
vector signed short vec_and (vector signed short, vector bool short);
vector signed short vec_and (vector signed short, vector signed short);
vector unsigned short vec_and (vector bool short,
                               vector unsigned short);
vector unsigned short vec_and (vector unsigned short,
                               vector bool short);
vector unsigned short vec_and (vector unsigned short,
                               vector unsigned short);
vector signed char vec_and (vector bool char, vector signed char);
vector bool char vec_and (vector bool char, vector bool char);
vector signed char vec_and (vector signed char, vector bool char);
vector signed char vec_and (vector signed char, vector signed char);
vector unsigned char vec_and (vector bool char, vector unsigned char);
vector unsigned char vec_and (vector unsigned char, vector bool 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 bool int);
vector float vec_andc (vector bool int, vector float);
vector bool int vec_andc (vector bool int, vector bool int);
vector signed int vec_andc (vector bool int, vector signed int);
vector signed int vec_andc (vector signed int, vector bool int);
vector signed int vec_andc (vector signed int, vector signed int);
vector unsigned int vec_andc (vector bool int, vector unsigned int);
vector unsigned int vec_andc (vector unsigned int, vector bool int);
vector unsigned int vec_andc (vector unsigned int, vector unsigned int);
vector bool short vec_andc (vector bool short, vector bool short);
vector signed short vec_andc (vector bool short, vector signed short);
vector signed short vec_andc (vector signed short, vector bool short);
vector signed short vec_andc (vector signed short, vector signed short);
vector unsigned short vec_andc (vector bool short,
                                vector unsigned short);
vector unsigned short vec_andc (vector unsigned short,
                                vector bool short);
vector unsigned short vec_andc (vector unsigned short,
                                vector unsigned short);
vector signed char vec_andc (vector bool char, vector signed char);
vector bool char vec_andc (vector bool char, vector bool char);
vector signed char vec_andc (vector signed char, vector bool char);
vector signed char vec_andc (vector signed char, vector signed char);
vector unsigned char vec_andc (vector bool char, vector unsigned char);
vector unsigned char vec_andc (vector unsigned char, vector bool 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 signed int vec_vavgsw (vector signed int, vector signed int);

vector unsigned int vec_vavguw (vector unsigned int,
                                vector unsigned int);

vector signed short vec_vavgsh (vector signed short,
                                vector signed short);

vector unsigned short vec_vavguh (vector unsigned short,
                                  vector unsigned short);

vector signed char vec_vavgsb (vector signed char, vector signed char);

vector unsigned char vec_vavgub (vector unsigned char,
                                 vector unsigned char);

vector float vec_ceil (vector float);

vector signed int vec_cmpb (vector float, vector float);

vector bool char vec_cmpeq (vector signed char, vector signed char);
vector bool char vec_cmpeq (vector unsigned char, vector unsigned char);
vector bool short vec_cmpeq (vector signed short, vector signed short);
vector bool short vec_cmpeq (vector unsigned short,
                             vector unsigned short);
vector bool int vec_cmpeq (vector signed int, vector signed int);
vector bool int vec_cmpeq (vector unsigned int, vector unsigned int);
vector bool int vec_cmpeq (vector float, vector float);

vector bool int vec_vcmpeqfp (vector float, vector float);

vector bool int vec_vcmpequw (vector signed int, vector signed int);
vector bool int vec_vcmpequw (vector unsigned int, vector unsigned int);

vector bool short vec_vcmpequh (vector signed short,
                                vector signed short);
vector bool short vec_vcmpequh (vector unsigned short,
                                vector unsigned short);

vector bool char vec_vcmpequb (vector signed char, vector signed char);
vector bool char vec_vcmpequb (vector unsigned char,
                               vector unsigned char);

vector bool int vec_cmpge (vector float, vector float);

vector bool char vec_cmpgt (vector unsigned char, vector unsigned char);
vector bool char vec_cmpgt (vector signed char, vector signed char);
vector bool short vec_cmpgt (vector unsigned short,
                             vector unsigned short);
vector bool short vec_cmpgt (vector signed short, vector signed short);
vector bool int vec_cmpgt (vector unsigned int, vector unsigned int);
vector bool int vec_cmpgt (vector signed int, vector signed int);
vector bool int vec_cmpgt (vector float, vector float);

vector bool int vec_vcmpgtfp (vector float, vector float);

vector bool int vec_vcmpgtsw (vector signed int, vector signed int);

vector bool int vec_vcmpgtuw (vector unsigned int, vector unsigned int);

vector bool short vec_vcmpgtsh (vector signed short,
                                vector signed short);

vector bool short vec_vcmpgtuh (vector unsigned short,
                                vector unsigned short);

vector bool char vec_vcmpgtsb (vector signed char, vector signed char);

vector bool char vec_vcmpgtub (vector unsigned char,
                               vector unsigned char);

vector bool int vec_cmple (vector float, vector float);

vector bool char vec_cmplt (vector unsigned char, vector unsigned char);
vector bool char vec_cmplt (vector signed char, vector signed char);
vector bool short vec_cmplt (vector unsigned short,
                             vector unsigned short);
vector bool short vec_cmplt (vector signed short, vector signed short);
vector bool int vec_cmplt (vector unsigned int, vector unsigned int);
vector bool int vec_cmplt (vector signed int, vector signed int);
vector bool int vec_cmplt (vector float, vector float);

vector float vec_ctf (vector unsigned int, const int);
vector float vec_ctf (vector signed int, const int);

vector float vec_vcfsx (vector signed int, const int);

vector float vec_vcfux (vector unsigned int, const int);

vector signed int vec_cts (vector float, const int);

vector unsigned int vec_ctu (vector float, const int);

void vec_dss (const int);
void vec_dssall (void);

void vec_dst (const vector unsigned char *, int, const int);
void vec_dst (const vector signed char *, int, const int);
void vec_dst (const vector bool char *, int, const int);
void vec_dst (const vector unsigned short *, int, const int);
void vec_dst (const vector signed short *, int, const int);
void vec_dst (const vector bool short *, int, const int);
void vec_dst (const vector pixel *, int, const int);
void vec_dst (const vector unsigned int *, int, const int);
void vec_dst (const vector signed int *, int, const int);
void vec_dst (const vector bool int *, int, const int);
void vec_dst (const vector float *, int, const int);
void vec_dst (const unsigned char *, int, const int);
void vec_dst (const signed char *, int, const int);
void vec_dst (const unsigned short *, int, const int);
void vec_dst (const short *, int, const int);
void vec_dst (const unsigned int *, int, const int);
void vec_dst (const int *, int, const int);
void vec_dst (const unsigned long *, int, const int);
void vec_dst (const long *, int, const int);
void vec_dst (const float *, int, const int);

void vec_dstst (const vector unsigned char *, int, const int);
void vec_dstst (const vector signed char *, int, const int);
void vec_dstst (const vector bool char *, int, const int);
void vec_dstst (const vector unsigned short *, int, const int);
void vec_dstst (const vector signed short *, int, const int);
void vec_dstst (const vector bool short *, int, const int);
void vec_dstst (const vector pixel *, int, const int);
void vec_dstst (const vector unsigned int *, int, const int);
void vec_dstst (const vector signed int *, int, const int);
void vec_dstst (const vector bool int *, int, const int);
void vec_dstst (const vector float *, int, const int);
void vec_dstst (const unsigned char *, int, const int);
void vec_dstst (const signed char *, int, const int);
void vec_dstst (const unsigned short *, int, const int);
void vec_dstst (const short *, int, const int);
void vec_dstst (const unsigned int *, int, const int);
void vec_dstst (const int *, int, const int);
void vec_dstst (const unsigned long *, int, const int);
void vec_dstst (const long *, int, const int);
void vec_dstst (const float *, int, const int);

void vec_dststt (const vector unsigned char *, int, const int);
void vec_dststt (const vector signed char *, int, const int);
void vec_dststt (const vector bool char *, int, const int);
void vec_dststt (const vector unsigned short *, int, const int);
void vec_dststt (const vector signed short *, int, const int);
void vec_dststt (const vector bool short *, int, const int);
void vec_dststt (const vector pixel *, int, const int);
void vec_dststt (const vector unsigned int *, int, const int);
void vec_dststt (const vector signed int *, int, const int);
void vec_dststt (const vector bool int *, int, const int);
void vec_dststt (const vector float *, int, const int);
void vec_dststt (const unsigned char *, int, const int);
void vec_dststt (const signed char *, int, const int);
void vec_dststt (const unsigned short *, int, const int);
void vec_dststt (const short *, int, const int);
void vec_dststt (const unsigned int *, int, const int);
void vec_dststt (const int *, int, const int);
void vec_dststt (const unsigned long *, int, const int);
void vec_dststt (const long *, int, const int);
void vec_dststt (const float *, int, const int);

void vec_dstt (const vector unsigned char *, int, const int);
void vec_dstt (const vector signed char *, int, const int);
void vec_dstt (const vector bool char *, int, const int);
void vec_dstt (const vector unsigned short *, int, const int);
void vec_dstt (const vector signed short *, int, const int);
void vec_dstt (const vector bool short *, int, const int);
void vec_dstt (const vector pixel *, int, const int);
void vec_dstt (const vector unsigned int *, int, const int);
void vec_dstt (const vector signed int *, int, const int);
void vec_dstt (const vector bool int *, int, const int);
void vec_dstt (const vector float *, int, const int);
void vec_dstt (const unsigned char *, int, const int);
void vec_dstt (const signed char *, int, const int);
void vec_dstt (const unsigned short *, int, const int);
void vec_dstt (const short *, int, const int);
void vec_dstt (const unsigned int *, int, const int);
void vec_dstt (const int *, int, const int);
void vec_dstt (const unsigned long *, int, const int);
void vec_dstt (const long *, int, const int);
void vec_dstt (const float *, int, const int);

vector float vec_expte (vector float);

vector float vec_floor (vector float);

vector float vec_ld (int, const vector float *);
vector float vec_ld (int, const float *);
vector bool int vec_ld (int, const vector bool int *);
vector signed int vec_ld (int, const vector signed int *);
vector signed int vec_ld (int, const int *);
vector signed int vec_ld (int, const long *);
vector unsigned int vec_ld (int, const vector unsigned int *);
vector unsigned int vec_ld (int, const unsigned int *);
vector unsigned int vec_ld (int, const unsigned long *);
vector bool short vec_ld (int, const vector bool short *);
vector pixel vec_ld (int, const vector pixel *);
vector signed short vec_ld (int, const vector signed short *);
vector signed short vec_ld (int, const short *);
vector unsigned short vec_ld (int, const vector unsigned short *);
vector unsigned short vec_ld (int, const unsigned short *);
vector bool char vec_ld (int, const vector bool char *);
vector signed char vec_ld (int, const vector signed char *);
vector signed char vec_ld (int, const signed char *);
vector unsigned char vec_ld (int, const vector unsigned char *);
vector unsigned char vec_ld (int, const unsigned char *);

vector signed char vec_lde (int, const signed char *);
vector unsigned char vec_lde (int, const unsigned char *);
vector signed short vec_lde (int, const short *);
vector unsigned short vec_lde (int, const unsigned short *);
vector float vec_lde (int, const float *);
vector signed int vec_lde (int, const int *);
vector unsigned int vec_lde (int, const unsigned int *);
vector signed int vec_lde (int, const long *);
vector unsigned int vec_lde (int, const unsigned long *);

vector float vec_lvewx (int, float *);
vector signed int vec_lvewx (int, int *);
vector unsigned int vec_lvewx (int, unsigned int *);
vector signed int vec_lvewx (int, long *);
vector unsigned int vec_lvewx (int, unsigned long *);

vector signed short vec_lvehx (int, short *);
vector unsigned short vec_lvehx (int, unsigned short *);

vector signed char vec_lvebx (int, char *);
vector unsigned char vec_lvebx (int, unsigned char *);

vector float vec_ldl (int, const vector float *);
vector float vec_ldl (int, const float *);
vector bool int vec_ldl (int, const vector bool int *);
vector signed int vec_ldl (int, const vector signed int *);
vector signed int vec_ldl (int, const int *);
vector signed int vec_ldl (int, const long *);
vector unsigned int vec_ldl (int, const vector unsigned int *);
vector unsigned int vec_ldl (int, const unsigned int *);
vector unsigned int vec_ldl (int, const unsigned long *);
vector bool short vec_ldl (int, const vector bool short *);
vector pixel vec_ldl (int, const vector pixel *);
vector signed short vec_ldl (int, const vector signed short *);
vector signed short vec_ldl (int, const short *);
vector unsigned short vec_ldl (int, const vector unsigned short *);
vector unsigned short vec_ldl (int, const unsigned short *);
vector bool char vec_ldl (int, const vector bool char *);
vector signed char vec_ldl (int, const vector signed char *);
vector signed char vec_ldl (int, const signed char *);
vector unsigned char vec_ldl (int, const vector unsigned char *);
vector unsigned char vec_ldl (int, const unsigned char *);

vector float vec_loge (vector float);

vector unsigned char vec_lvsl (int, const volatile unsigned char *);
vector unsigned char vec_lvsl (int, const volatile signed char *);
vector unsigned char vec_lvsl (int, const volatile unsigned short *);
vector unsigned char vec_lvsl (int, const volatile short *);
vector unsigned char vec_lvsl (int, const volatile unsigned int *);
vector unsigned char vec_lvsl (int, const volatile int *);
vector unsigned char vec_lvsl (int, const volatile unsigned long *);
vector unsigned char vec_lvsl (int, const volatile long *);
vector unsigned char vec_lvsl (int, const volatile float *);

vector unsigned char vec_lvsr (int, const volatile unsigned char *);
vector unsigned char vec_lvsr (int, const volatile signed char *);
vector unsigned char vec_lvsr (int, const volatile unsigned short *);
vector unsigned char vec_lvsr (int, const volatile short *);
vector unsigned char vec_lvsr (int, const volatile unsigned int *);
vector unsigned char vec_lvsr (int, const volatile int *);
vector unsigned char vec_lvsr (int, const volatile unsigned long *);
vector unsigned char vec_lvsr (int, const volatile long *);
vector unsigned char vec_lvsr (int, const volatile float *);

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 bool char, vector unsigned char);
vector unsigned char vec_max (vector unsigned char, vector bool char);
vector unsigned char vec_max (vector unsigned char,
                              vector unsigned char);
vector signed char vec_max (vector bool char, vector signed char);
vector signed char vec_max (vector signed char, vector bool char);
vector signed char vec_max (vector signed char, vector signed char);
vector unsigned short vec_max (vector bool short,
                               vector unsigned short);
vector unsigned short vec_max (vector unsigned short,
                               vector bool short);
vector unsigned short vec_max (vector unsigned short,
                               vector unsigned short);
vector signed short vec_max (vector bool short, vector signed short);
vector signed short vec_max (vector signed short, vector bool short);
vector signed short vec_max (vector signed short, vector signed short);
vector unsigned int vec_max (vector bool int, vector unsigned int);
vector unsigned int vec_max (vector unsigned int, vector bool int);
vector unsigned int vec_max (vector unsigned int, vector unsigned int);
vector signed int vec_max (vector bool int, vector signed int);
vector signed int vec_max (vector signed int, vector bool int);
vector signed int vec_max (vector signed int, vector signed int);
vector float vec_max (vector float, vector float);

vector float vec_vmaxfp (vector float, vector float);

vector signed int vec_vmaxsw (vector bool int, vector signed int);
vector signed int vec_vmaxsw (vector signed int, vector bool int);
vector signed int vec_vmaxsw (vector signed int, vector signed int);

vector unsigned int vec_vmaxuw (vector bool int, vector unsigned int);
vector unsigned int vec_vmaxuw (vector unsigned int, vector bool int);
vector unsigned int vec_vmaxuw (vector unsigned int,
                                vector unsigned int);

vector signed short vec_vmaxsh (vector bool short, vector signed short);
vector signed short vec_vmaxsh (vector signed short, vector bool short);
vector signed short vec_vmaxsh (vector signed short,
                                vector signed short);

vector unsigned short vec_vmaxuh (vector bool short,
                                  vector unsigned short);
vector unsigned short vec_vmaxuh (vector unsigned short,
                                  vector bool short);
vector unsigned short vec_vmaxuh (vector unsigned short,
                                  vector unsigned short);

vector signed char vec_vmaxsb (vector bool char, vector signed char);
vector signed char vec_vmaxsb (vector signed char, vector bool char);
vector signed char vec_vmaxsb (vector signed char, vector signed char);

vector unsigned char vec_vmaxub (vector bool char,
                                 vector unsigned char);
vector unsigned char vec_vmaxub (vector unsigned char,
                                 vector bool char);
vector unsigned char vec_vmaxub (vector unsigned char,
                                 vector unsigned char);

vector bool char vec_mergeh (vector bool char, vector bool char);
vector signed char vec_mergeh (vector signed char, vector signed char);
vector unsigned char vec_mergeh (vector unsigned char,
                                 vector unsigned char);
vector bool short vec_mergeh (vector bool short, vector bool short);
vector pixel vec_mergeh (vector pixel, vector pixel);
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 bool int vec_mergeh (vector bool int, vector bool int);
vector signed int vec_mergeh (vector signed int, vector signed int);
vector unsigned int vec_mergeh (vector unsigned int,
                                vector unsigned int);

vector float vec_vmrghw (vector float, vector float);
vector bool int vec_vmrghw (vector bool int, vector bool int);
vector signed int vec_vmrghw (vector signed int, vector signed int);
vector unsigned int vec_vmrghw (vector unsigned int,
                                vector unsigned int);

vector bool short vec_vmrghh (vector bool short, vector bool short);
vector signed short vec_vmrghh (vector signed short,
                                vector signed short);
vector unsigned short vec_vmrghh (vector unsigned short,
                                  vector unsigned short);
vector pixel vec_vmrghh (vector pixel, vector pixel);

vector bool char vec_vmrghb (vector bool char, vector bool char);
vector signed char vec_vmrghb (vector signed char, vector signed char);
vector unsigned char vec_vmrghb (vector unsigned char,
                                 vector unsigned char);

vector bool char vec_mergel (vector bool char, vector bool char);
vector signed char vec_mergel (vector signed char, vector signed char);
vector unsigned char vec_mergel (vector unsigned char,
                                 vector unsigned char);
vector bool short vec_mergel (vector bool short, vector bool short);
vector pixel vec_mergel (vector pixel, vector pixel);
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 bool int vec_mergel (vector bool int, vector bool int);
vector signed int vec_mergel (vector signed int, vector signed int);
vector unsigned int vec_mergel (vector unsigned int,
                                vector unsigned int);

vector float vec_vmrglw (vector float, vector float);
vector signed int vec_vmrglw (vector signed int, vector signed int);
vector unsigned int vec_vmrglw (vector unsigned int,
                                vector unsigned int);
vector bool int vec_vmrglw (vector bool int, vector bool int);

vector bool short vec_vmrglh (vector bool short, vector bool short);
vector signed short vec_vmrglh (vector signed short,
                                vector signed short);
vector unsigned short vec_vmrglh (vector unsigned short,
                                  vector unsigned short);
vector pixel vec_vmrglh (vector pixel, vector pixel);

vector bool char vec_vmrglb (vector bool char, vector bool char);
vector signed char vec_vmrglb (vector signed char, vector signed char);
vector unsigned char vec_vmrglb (vector unsigned char,
                                 vector unsigned char);

vector unsigned short vec_mfvscr (void);
vector unsigned char vec_min (vector bool char, vector unsigned char);
vector unsigned char vec_min (vector unsigned char, vector bool char);
vector unsigned char vec_min (vector unsigned char,
                              vector unsigned char);
vector signed char vec_min (vector bool char, vector signed char);
vector signed char vec_min (vector signed char, vector bool char);
vector signed char vec_min (vector signed char, vector signed char);
vector unsigned short vec_min (vector bool short,
                               vector unsigned short);
vector unsigned short vec_min (vector unsigned short,
                               vector bool short);
vector unsigned short vec_min (vector unsigned short,
                               vector unsigned short);
vector signed short vec_min (vector bool short, vector signed short);
vector signed short vec_min (vector signed short, vector bool short);
vector signed short vec_min (vector signed short, vector signed short);
vector unsigned int vec_min (vector bool int, vector unsigned int);
vector unsigned int vec_min (vector unsigned int, vector bool int);
vector unsigned int vec_min (vector unsigned int, vector unsigned int);
vector signed int vec_min (vector bool int, vector signed int);
vector signed int vec_min (vector signed int, vector bool int);
vector signed int vec_min (vector signed int, vector signed int);
vector float vec_min (vector float, vector float);

vector float vec_vminfp (vector float, vector float);

vector signed int vec_vminsw (vector bool int, vector signed int);
vector signed int vec_vminsw (vector signed int, vector bool int);
vector signed int vec_vminsw (vector signed int, vector signed int);

vector unsigned int vec_vminuw (vector bool int, vector unsigned int);
vector unsigned int vec_vminuw (vector unsigned int, vector bool int);
vector unsigned int vec_vminuw (vector unsigned int,
                                vector unsigned int);

vector signed short vec_vminsh (vector bool short, vector signed short);
vector signed short vec_vminsh (vector signed short, vector bool short);
vector signed short vec_vminsh (vector signed short,
                                vector signed short);

vector unsigned short vec_vminuh (vector bool short,
                                  vector unsigned short);
vector unsigned short vec_vminuh (vector unsigned short,
                                  vector bool short);
vector unsigned short vec_vminuh (vector unsigned short,
                                  vector unsigned short);

vector signed char vec_vminsb (vector bool char, vector signed char);
vector signed char vec_vminsb (vector signed char, vector bool char);
vector signed char vec_vminsb (vector signed char, vector signed char);

vector unsigned char vec_vminub (vector bool char,
                                 vector unsigned char);
vector unsigned char vec_vminub (vector unsigned char,
                                 vector bool char);
vector unsigned char vec_vminub (vector unsigned char,
                                 vector unsigned char);

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 signed int vec_vmsumshm (vector signed short,
                                vector signed short,
                                vector signed int);

vector unsigned int vec_vmsumuhm (vector unsigned short,
                                  vector unsigned short,
                                  vector unsigned int);

vector signed int vec_vmsummbm (vector signed char,
                                vector unsigned char,
                                vector signed int);

vector unsigned int vec_vmsumubm (vector unsigned char,
                                  vector unsigned char,
                                  vector unsigned 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);

vector signed int vec_vmsumshs (vector signed short,
                                vector signed short,
                                vector signed int);

vector unsigned int vec_vmsumuhs (vector unsigned short,
                                  vector unsigned short,
                                  vector unsigned int);

void vec_mtvscr (vector signed int);
void vec_mtvscr (vector unsigned int);
void vec_mtvscr (vector bool int);
void vec_mtvscr (vector signed short);
void vec_mtvscr (vector unsigned short);
void vec_mtvscr (vector bool short);
void vec_mtvscr (vector pixel);
void vec_mtvscr (vector signed char);
void vec_mtvscr (vector unsigned char);
void vec_mtvscr (vector bool 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 signed int vec_vmulesh (vector signed short,
                               vector signed short);

vector unsigned int vec_vmuleuh (vector unsigned short,
                                 vector unsigned short);

vector signed short vec_vmulesb (vector signed char,
                                 vector signed char);

vector unsigned short vec_vmuleub (vector unsigned char,
                                   vector unsigned char);

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 signed int vec_vmulosh (vector signed short,
                               vector signed short);

vector unsigned int vec_vmulouh (vector unsigned short,
                                 vector unsigned short);

vector signed short vec_vmulosb (vector signed char,
                                 vector signed char);

vector unsigned short vec_vmuloub (vector unsigned char,
                                    vector unsigned char);

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 bool int vec_nor (vector bool int, vector bool int);
vector signed short vec_nor (vector signed short, vector signed short);
vector unsigned short vec_nor (vector unsigned short,
                               vector unsigned short);
vector bool short vec_nor (vector bool short, vector bool short);
vector signed char vec_nor (vector signed char, vector signed char);
vector unsigned char vec_nor (vector unsigned char,
                              vector unsigned char);
vector bool char vec_nor (vector bool char, vector bool char);

vector float vec_or (vector float, vector float);
vector float vec_or (vector float, vector bool int);
vector float vec_or (vector bool int, vector float);
vector bool int vec_or (vector bool int, vector bool int);
vector signed int vec_or (vector bool int, vector signed int);
vector signed int vec_or (vector signed int, vector bool int);
vector signed int vec_or (vector signed int, vector signed int);
vector unsigned int vec_or (vector bool int, vector unsigned int);
vector unsigned int vec_or (vector unsigned int, vector bool int);
vector unsigned int vec_or (vector unsigned int, vector unsigned int);
vector bool short vec_or (vector bool short, vector bool short);
vector signed short vec_or (vector bool short, vector signed short);
vector signed short vec_or (vector signed short, vector bool short);
vector signed short vec_or (vector signed short, vector signed short);
vector unsigned short vec_or (vector bool short, vector unsigned short);
vector unsigned short vec_or (vector unsigned short, vector bool short);
vector unsigned short vec_or (vector unsigned short,
                              vector unsigned short);
vector signed char vec_or (vector bool char, vector signed char);
vector bool char vec_or (vector bool char, vector bool char);
vector signed char vec_or (vector signed char, vector bool char);
vector signed char vec_or (vector signed char, vector signed char);
vector unsigned char vec_or (vector bool char, vector unsigned char);
vector unsigned char vec_or (vector unsigned char, vector bool 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 bool char vec_pack (vector bool short, vector bool short);
vector signed short vec_pack (vector signed int, vector signed int);
vector unsigned short vec_pack (vector unsigned int,
                                vector unsigned int);
vector bool short vec_pack (vector bool int, vector bool int);

vector bool short vec_vpkuwum (vector bool int, vector bool int);
vector signed short vec_vpkuwum (vector signed int, vector signed int);
vector unsigned short vec_vpkuwum (vector unsigned int,
                                   vector unsigned int);

vector bool char vec_vpkuhum (vector bool short, vector bool short);
vector signed char vec_vpkuhum (vector signed short,
                                vector signed short);
vector unsigned char vec_vpkuhum (vector unsigned short,
                                  vector unsigned short);

vector pixel 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 signed short vec_vpkswss (vector signed int, vector signed int);

vector unsigned short vec_vpkuwus (vector unsigned int,
                                   vector unsigned int);

vector signed char vec_vpkshss (vector signed short,
                                vector signed short);
vector unsigned char vec_vpkuhus (vector unsigned short,
                                  vector unsigned short);

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 unsigned short vec_vpkswus (vector signed int,
                                   vector signed int);

vector unsigned char vec_vpkshus (vector signed short,
                                  vector signed short);

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 bool int vec_perm (vector bool int,
                          vector bool 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 bool short vec_perm (vector bool short,
                            vector bool short,
                            vector unsigned char);
vector pixel vec_perm (vector pixel,
                       vector pixel,
                       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 bool char vec_perm (vector bool char,
                           vector bool 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 signed int vec_vrlw (vector signed int, vector unsigned int);
vector unsigned int vec_vrlw (vector unsigned int, vector unsigned int);

vector signed short vec_vrlh (vector signed short,
                              vector unsigned short);
vector unsigned short vec_vrlh (vector unsigned short,
                                vector unsigned short);

vector signed char vec_vrlb (vector signed char, vector unsigned char);
vector unsigned char vec_vrlb (vector unsigned char,
                               vector unsigned char);

vector float vec_round (vector float);

vector float vec_rsqrte (vector float);

vector float vec_sel (vector float, vector float, vector bool int);
vector float vec_sel (vector float, vector float, vector unsigned int);
vector signed int vec_sel (vector signed int,
                           vector signed int,
                           vector bool 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 bool int);
vector unsigned int vec_sel (vector unsigned int,
                             vector unsigned int,
                             vector unsigned int);
vector bool int vec_sel (vector bool int,
                         vector bool int,
                         vector bool int);
vector bool int vec_sel (vector bool int,
                         vector bool int,
                         vector unsigned int);
vector signed short vec_sel (vector signed short,
                             vector signed short,
                             vector bool 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 bool short);
vector unsigned short vec_sel (vector unsigned short,
                               vector unsigned short,
                               vector unsigned short);
vector bool short vec_sel (vector bool short,
                           vector bool short,
                           vector bool short);
vector bool short vec_sel (vector bool short,
                           vector bool short,
                           vector unsigned short);
vector signed char vec_sel (vector signed char,
                            vector signed char,
                            vector bool 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 bool char);
vector unsigned char vec_sel (vector unsigned char,
                              vector unsigned char,
                              vector unsigned char);
vector bool char vec_sel (vector bool char,
                          vector bool char,
                          vector bool char);
vector bool char vec_sel (vector bool char,
                          vector bool 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 signed int vec_vslw (vector signed int, vector unsigned int);
vector unsigned int vec_vslw (vector unsigned int, vector unsigned int);

vector signed short vec_vslh (vector signed short,
                              vector unsigned short);
vector unsigned short vec_vslh (vector unsigned short,
                                vector unsigned short);

vector signed char vec_vslb (vector signed char, vector unsigned char);
vector unsigned char vec_vslb (vector unsigned char,
                               vector unsigned char);

vector float vec_sld (vector float, vector float, const int);
vector signed int vec_sld (vector signed int,
                           vector signed int,
                           const int);
vector unsigned int vec_sld (vector unsigned int,
                             vector unsigned int,
                             const int);
vector bool int vec_sld (vector bool int,
                         vector bool int,
                         const int);
vector signed short vec_sld (vector signed short,
                             vector signed short,
                             const int);
vector unsigned short vec_sld (vector unsigned short,
                               vector unsigned short,
                               const int);
vector bool short vec_sld (vector bool short,
                           vector bool short,
                           const int);
vector pixel vec_sld (vector pixel,
                      vector pixel,
                      const int);
vector signed char vec_sld (vector signed char,
                            vector signed char,
                            const int);
vector unsigned char vec_sld (vector unsigned char,
                              vector unsigned char,
                              const int);
vector bool char vec_sld (vector bool char,
                          vector bool char,
                          const int);

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 bool int vec_sll (vector bool int,
                         vector unsigned int);
vector bool int vec_sll (vector bool int,
                         vector unsigned short);
vector bool int vec_sll (vector bool 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 bool short vec_sll (vector bool short, vector unsigned int);
vector bool short vec_sll (vector bool short, vector unsigned short);
vector bool short vec_sll (vector bool short, vector unsigned char);
vector pixel vec_sll (vector pixel, vector unsigned int);
vector pixel vec_sll (vector pixel, vector unsigned short);
vector pixel vec_sll (vector pixel, 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 bool char vec_sll (vector bool char, vector unsigned int);
vector bool char vec_sll (vector bool char, vector unsigned short);
vector bool char vec_sll (vector bool 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 pixel vec_slo (vector pixel, vector signed char);
vector pixel vec_slo (vector pixel, 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 int);
vector unsigned char vec_splat (vector unsigned char, const int);
vector bool char vec_splat (vector bool char, const int);
vector signed short vec_splat (vector signed short, const int);
vector unsigned short vec_splat (vector unsigned short, const int);
vector bool short vec_splat (vector bool short, const int);
vector pixel vec_splat (vector pixel, const int);
vector float vec_splat (vector float, const int);
vector signed int vec_splat (vector signed int, const int);
vector unsigned int vec_splat (vector unsigned int, const int);
vector bool int vec_splat (vector bool int, const int);

vector float vec_vspltw (vector float, const int);
vector signed int vec_vspltw (vector signed int, const int);
vector unsigned int vec_vspltw (vector unsigned int, const int);
vector bool int vec_vspltw (vector bool int, const int);

vector bool short vec_vsplth (vector bool short, const int);
vector signed short vec_vsplth (vector signed short, const int);
vector unsigned short vec_vsplth (vector unsigned short, const int);
vector pixel vec_vsplth (vector pixel, const int);

vector signed char vec_vspltb (vector signed char, const int);
vector unsigned char vec_vspltb (vector unsigned char, const int);
vector bool char vec_vspltb (vector bool char, const int);

vector signed char vec_splat_s8 (const int);

vector signed short vec_splat_s16 (const int);

vector signed int vec_splat_s32 (const int);

vector unsigned char vec_splat_u8 (const int);

vector unsigned short vec_splat_u16 (const int);

vector unsigned int vec_splat_u32 (const int);

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 int vec_vsrw (vector signed int, vector unsigned int);
vector unsigned int vec_vsrw (vector unsigned int, vector unsigned int);
vector signed short vec_vsrh (vector signed short,
                              vector unsigned short);
vector unsigned short vec_vsrh (vector unsigned short,
                                vector unsigned short);

vector signed char vec_vsrb (vector signed char, vector unsigned char);
vector unsigned char vec_vsrb (vector unsigned char,
                               vector unsigned char);

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_vsraw (vector signed int, vector unsigned int);
vector unsigned int vec_vsraw (vector unsigned int,
                               vector unsigned int);

vector signed short vec_vsrah (vector signed short,
                               vector unsigned short);
vector unsigned short vec_vsrah (vector unsigned short,
                                 vector unsigned short);

vector signed char vec_vsrab (vector signed char, vector unsigned char);
vector unsigned char vec_vsrab (vector unsigned char,
                                vector unsigned char);

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 bool int vec_srl (vector bool int, vector unsigned int);
vector bool int vec_srl (vector bool int, vector unsigned short);
vector bool int vec_srl (vector bool 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 bool short vec_srl (vector bool short, vector unsigned int);
vector bool short vec_srl (vector bool short, vector unsigned short);
vector bool short vec_srl (vector bool short, vector unsigned char);
vector pixel vec_srl (vector pixel, vector unsigned int);
vector pixel vec_srl (vector pixel, vector unsigned short);
vector pixel vec_srl (vector pixel, 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 bool char vec_srl (vector bool char, vector unsigned int);
vector bool char vec_srl (vector bool char, vector unsigned short);
vector bool char vec_srl (vector bool 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 pixel vec_sro (vector pixel, vector signed char);
vector pixel vec_sro (vector pixel, 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, vector float *);
void vec_st (vector float, int, float *);
void vec_st (vector signed int, int, vector signed int *);
void vec_st (vector signed int, int, int *);
void vec_st (vector unsigned int, int, vector unsigned int *);
void vec_st (vector unsigned int, int, unsigned int *);
void vec_st (vector bool int, int, vector bool int *);
void vec_st (vector bool int, int, unsigned int *);
void vec_st (vector bool int, int, int *);
void vec_st (vector signed short, int, vector signed short *);
void vec_st (vector signed short, int, short *);
void vec_st (vector unsigned short, int, vector unsigned short *);
void vec_st (vector unsigned short, int, unsigned short *);
void vec_st (vector bool short, int, vector bool short *);
void vec_st (vector bool short, int, unsigned short *);
void vec_st (vector pixel, int, vector pixel *);
void vec_st (vector pixel, int, unsigned short *);
void vec_st (vector pixel, int, short *);
void vec_st (vector bool short, int, short *);
void vec_st (vector signed char, int, vector signed char *);
void vec_st (vector signed char, int, signed char *);
void vec_st (vector unsigned char, int, vector unsigned char *);
void vec_st (vector unsigned char, int, unsigned char *);
void vec_st (vector bool char, int, vector bool char *);
void vec_st (vector bool char, int, unsigned char *);
void vec_st (vector bool char, int, signed char *);

void vec_ste (vector signed char, int, signed char *);
void vec_ste (vector unsigned char, int, unsigned char *);
void vec_ste (vector bool char, int, signed char *);
void vec_ste (vector bool char, int, unsigned char *);
void vec_ste (vector signed short, int, short *);
void vec_ste (vector unsigned short, int, unsigned short *);
void vec_ste (vector bool short, int, short *);
void vec_ste (vector bool short, int, unsigned short *);
void vec_ste (vector pixel, int, short *);
void vec_ste (vector pixel, int, unsigned short *);
void vec_ste (vector float, int, float *);
void vec_ste (vector signed int, int, int *);
void vec_ste (vector unsigned int, int, unsigned int *);
void vec_ste (vector bool int, int, int *);
void vec_ste (vector bool int, int, unsigned int *);

void vec_stvewx (vector float, int, float *);
void vec_stvewx (vector signed int, int, int *);
void vec_stvewx (vector unsigned int, int, unsigned int *);
void vec_stvewx (vector bool int, int, int *);
void vec_stvewx (vector bool int, int, unsigned int *);

void vec_stvehx (vector signed short, int, short *);
void vec_stvehx (vector unsigned short, int, unsigned short *);
void vec_stvehx (vector bool short, int, short *);
void vec_stvehx (vector bool short, int, unsigned short *);
void vec_stvehx (vector pixel, int, short *);
void vec_stvehx (vector pixel, int, unsigned short *);

void vec_stvebx (vector signed char, int, signed char *);
void vec_stvebx (vector unsigned char, int, unsigned char *);
void vec_stvebx (vector bool char, int, signed char *);
void vec_stvebx (vector bool char, int, unsigned char *);

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 unsigned int, int, vector unsigned int *);
void vec_stl (vector unsigned int, int, unsigned int *);
void vec_stl (vector bool int, int, vector bool int *);
void vec_stl (vector bool int, int, unsigned int *);
void vec_stl (vector bool int, int, int *);
void vec_stl (vector signed short, int, vector signed short *);
void vec_stl (vector signed short, int, short *);
void vec_stl (vector unsigned short, int, vector unsigned short *);
void vec_stl (vector unsigned short, int, unsigned short *);
void vec_stl (vector bool short, int, vector bool short *);
void vec_stl (vector bool short, int, unsigned short *);
void vec_stl (vector bool short, int, short *);
void vec_stl (vector pixel, int, vector pixel *);
void vec_stl (vector pixel, int, unsigned short *);
void vec_stl (vector pixel, int, short *);
void vec_stl (vector signed char, int, vector signed char *);
void vec_stl (vector signed char, int, signed char *);
void vec_stl (vector unsigned char, int, vector unsigned char *);
void vec_stl (vector unsigned char, int, unsigned char *);
void vec_stl (vector bool char, int, vector bool char *);
void vec_stl (vector bool char, int, unsigned char *);
void vec_stl (vector bool char, int, signed char *);

vector signed char vec_sub (vector bool char, vector signed char);
vector signed char vec_sub (vector signed char, vector bool char);
vector signed char vec_sub (vector signed char, vector signed char);
vector unsigned char vec_sub (vector bool char, vector unsigned char);
vector unsigned char vec_sub (vector unsigned char, vector bool char);
vector unsigned char vec_sub (vector unsigned char,
                              vector unsigned char);
vector signed short vec_sub (vector bool short, vector signed short);
vector signed short vec_sub (vector signed short, vector bool short);
vector signed short vec_sub (vector signed short, vector signed short);
vector unsigned short vec_sub (vector bool short,
                               vector unsigned short);
vector unsigned short vec_sub (vector unsigned short,
                               vector bool short);
vector unsigned short vec_sub (vector unsigned short,
                               vector unsigned short);
vector signed int vec_sub (vector bool int, vector signed int);
vector signed int vec_sub (vector signed int, vector bool int);
vector signed int vec_sub (vector signed int, vector signed int);
vector unsigned int vec_sub (vector bool int, vector unsigned int);
vector unsigned int vec_sub (vector unsigned int, vector bool int);
vector unsigned int vec_sub (vector unsigned int, vector unsigned int);
vector float vec_sub (vector float, vector float);

vector float vec_vsubfp (vector float, vector float);

vector signed int vec_vsubuwm (vector bool int, vector signed int);
vector signed int vec_vsubuwm (vector signed int, vector bool int);
vector signed int vec_vsubuwm (vector signed int, vector signed int);
vector unsigned int vec_vsubuwm (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuwm (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuwm (vector unsigned int,
                                 vector unsigned int);

vector signed short vec_vsubuhm (vector bool short,
                                 vector signed short);
vector signed short vec_vsubuhm (vector signed short,
                                 vector bool short);
vector signed short vec_vsubuhm (vector signed short,
                                 vector signed short);
vector unsigned short vec_vsubuhm (vector bool short,
                                   vector unsigned short);
vector unsigned short vec_vsubuhm (vector unsigned short,
                                   vector bool short);
vector unsigned short vec_vsubuhm (vector unsigned short,
                                   vector unsigned short);

vector signed char vec_vsububm (vector bool char, vector signed char);
vector signed char vec_vsububm (vector signed char, vector bool char);
vector signed char vec_vsububm (vector signed char, vector signed char);
vector unsigned char vec_vsububm (vector bool char,
                                  vector unsigned char);
vector unsigned char vec_vsububm (vector unsigned char,
                                  vector bool char);
vector unsigned char vec_vsububm (vector unsigned char,
                                  vector unsigned char);

vector unsigned int vec_subc (vector unsigned int, vector unsigned int);

vector unsigned char vec_subs (vector bool char, vector unsigned char);
vector unsigned char vec_subs (vector unsigned char, vector bool char);
vector unsigned char vec_subs (vector unsigned char,
                               vector unsigned char);
vector signed char vec_subs (vector bool char, vector signed char);
vector signed char vec_subs (vector signed char, vector bool char);
vector signed char vec_subs (vector signed char, vector signed char);
vector unsigned short vec_subs (vector bool short,
                                vector unsigned short);
vector unsigned short vec_subs (vector unsigned short,
                                vector bool short);
vector unsigned short vec_subs (vector unsigned short,
                                vector unsigned short);
vector signed short vec_subs (vector bool short, vector signed short);
vector signed short vec_subs (vector signed short, vector bool short);
vector signed short vec_subs (vector signed short, vector signed short);
vector unsigned int vec_subs (vector bool int, vector unsigned int);
vector unsigned int vec_subs (vector unsigned int, vector bool int);
vector unsigned int vec_subs (vector unsigned int, vector unsigned int);
vector signed int vec_subs (vector bool int, vector signed int);
vector signed int vec_subs (vector signed int, vector bool int);
vector signed int vec_subs (vector signed int, vector signed int);

vector signed int vec_vsubsws (vector bool int, vector signed int);
vector signed int vec_vsubsws (vector signed int, vector bool int);
vector signed int vec_vsubsws (vector signed int, vector signed int);

vector unsigned int vec_vsubuws (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuws (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuws (vector unsigned int,
                                 vector unsigned int);

vector signed short vec_vsubshs (vector bool short,
                                 vector signed short);
vector signed short vec_vsubshs (vector signed short,
                                 vector bool short);
vector signed short vec_vsubshs (vector signed short,
                                 vector signed short);

vector unsigned short vec_vsubuhs (vector bool short,
                                   vector unsigned short);
vector unsigned short vec_vsubuhs (vector unsigned short,
                                   vector bool short);
vector unsigned short vec_vsubuhs (vector unsigned short,
                                   vector unsigned short);

vector signed char vec_vsubsbs (vector bool char, vector signed char);
vector signed char vec_vsubsbs (vector signed char, vector bool char);
vector signed char vec_vsubsbs (vector signed char, vector signed char);

vector unsigned char vec_vsububs (vector bool char,
                                  vector unsigned char);
vector unsigned char vec_vsububs (vector unsigned char,
                                  vector bool char);
vector unsigned char vec_vsububs (vector unsigned char,
                                  vector unsigned char);

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_vsum4shs (vector signed short, vector signed int);

vector signed int vec_vsum4sbs (vector signed char, vector signed int);

vector unsigned int vec_vsum4ubs (vector unsigned char,
                                  vector unsigned 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 bool short vec_unpackh (vector bool char);
vector signed int vec_unpackh (vector signed short);
vector bool int vec_unpackh (vector bool short);
vector unsigned int vec_unpackh (vector pixel);

vector bool int vec_vupkhsh (vector bool short);
vector signed int vec_vupkhsh (vector signed short);

vector unsigned int vec_vupkhpx (vector pixel);

vector bool short vec_vupkhsb (vector bool char);
vector signed short vec_vupkhsb (vector signed char);

vector signed short vec_unpackl (vector signed char);
vector bool short vec_unpackl (vector bool char);
vector unsigned int vec_unpackl (vector pixel);
vector signed int vec_unpackl (vector signed short);
vector bool int vec_unpackl (vector bool short);

vector unsigned int vec_vupklpx (vector pixel);

vector bool int vec_vupklsh (vector bool short);
vector signed int vec_vupklsh (vector signed short);

vector bool short vec_vupklsb (vector bool char);
vector signed short vec_vupklsb (vector signed char);

vector float vec_xor (vector float, vector float);
vector float vec_xor (vector float, vector bool int);
vector float vec_xor (vector bool int, vector float);
vector bool int vec_xor (vector bool int, vector bool int);
vector signed int vec_xor (vector bool int, vector signed int);
vector signed int vec_xor (vector signed int, vector bool int);
vector signed int vec_xor (vector signed int, vector signed int);
vector unsigned int vec_xor (vector bool int, vector unsigned int);
vector unsigned int vec_xor (vector unsigned int, vector bool int);
vector unsigned int vec_xor (vector unsigned int, vector unsigned int);
vector bool short vec_xor (vector bool short, vector bool short);
vector signed short vec_xor (vector bool short, vector signed short);
vector signed short vec_xor (vector signed short, vector bool short);
vector signed short vec_xor (vector signed short, vector signed short);
vector unsigned short vec_xor (vector bool short,
                               vector unsigned short);
vector unsigned short vec_xor (vector unsigned short,
                               vector bool short);
vector unsigned short vec_xor (vector unsigned short,
                               vector unsigned short);
vector signed char vec_xor (vector bool char, vector signed char);
vector bool char vec_xor (vector bool char, vector bool char);
vector signed char vec_xor (vector signed char, vector bool char);
vector signed char vec_xor (vector signed char, vector signed char);
vector unsigned char vec_xor (vector bool char, vector unsigned char);
vector unsigned char vec_xor (vector unsigned char, vector bool char);
vector unsigned char vec_xor (vector unsigned char,
                              vector unsigned char);

int vec_all_eq (vector signed char, vector bool char);
int vec_all_eq (vector signed char, vector signed char);
int vec_all_eq (vector unsigned char, vector bool char);
int vec_all_eq (vector unsigned char, vector unsigned char);
int vec_all_eq (vector bool char, vector bool char);
int vec_all_eq (vector bool char, vector unsigned char);
int vec_all_eq (vector bool char, vector signed char);
int vec_all_eq (vector signed short, vector bool short);
int vec_all_eq (vector signed short, vector signed short);
int vec_all_eq (vector unsigned short, vector bool short);
int vec_all_eq (vector unsigned short, vector unsigned short);
int vec_all_eq (vector bool short, vector bool short);
int vec_all_eq (vector bool short, vector unsigned short);
int vec_all_eq (vector bool short, vector signed short);
int vec_all_eq (vector pixel, vector pixel);
int vec_all_eq (vector signed int, vector bool int);
int vec_all_eq (vector signed int, vector signed int);
int vec_all_eq (vector unsigned int, vector bool int);
int vec_all_eq (vector unsigned int, vector unsigned int);
int vec_all_eq (vector bool int, vector bool int);
int vec_all_eq (vector bool int, vector unsigned int);
int vec_all_eq (vector bool int, vector signed int);
int vec_all_eq (vector float, vector float);

int vec_all_ge (vector bool char, vector unsigned char);
int vec_all_ge (vector unsigned char, vector bool char);
int vec_all_ge (vector unsigned char, vector unsigned char);
int vec_all_ge (vector bool char, vector signed char);
int vec_all_ge (vector signed char, vector bool char);
int vec_all_ge (vector signed char, vector signed char);
int vec_all_ge (vector bool short, vector unsigned short);
int vec_all_ge (vector unsigned short, vector bool short);
int vec_all_ge (vector unsigned short, vector unsigned short);
int vec_all_ge (vector signed short, vector signed short);
int vec_all_ge (vector bool short, vector signed short);
int vec_all_ge (vector signed short, vector bool short);
int vec_all_ge (vector bool int, vector unsigned int);
int vec_all_ge (vector unsigned int, vector bool int);
int vec_all_ge (vector unsigned int, vector unsigned int);
int vec_all_ge (vector bool int, vector signed int);
int vec_all_ge (vector signed int, vector bool int);
int vec_all_ge (vector signed int, vector signed int);
int vec_all_ge (vector float, vector float);

int vec_all_gt (vector bool char, vector unsigned char);
int vec_all_gt (vector unsigned char, vector bool char);
int vec_all_gt (vector unsigned char, vector unsigned char);
int vec_all_gt (vector bool char, vector signed char);
int vec_all_gt (vector signed char, vector bool char);
int vec_all_gt (vector signed char, vector signed char);
int vec_all_gt (vector bool short, vector unsigned short);
int vec_all_gt (vector unsigned short, vector bool short);
int vec_all_gt (vector unsigned short, vector unsigned short);
int vec_all_gt (vector bool short, vector signed short);
int vec_all_gt (vector signed short, vector bool short);
int vec_all_gt (vector signed short, vector signed short);
int vec_all_gt (vector bool int, vector unsigned int);
int vec_all_gt (vector unsigned int, vector bool int);
int vec_all_gt (vector unsigned int, vector unsigned int);
int vec_all_gt (vector bool int, vector signed int);
int vec_all_gt (vector signed int, vector bool int);
int vec_all_gt (vector signed int, vector signed int);
int vec_all_gt (vector float, vector float);
int vec_all_in (vector float, vector float);

int vec_all_le (vector bool char, vector unsigned char);
int vec_all_le (vector unsigned char, vector bool char);
int vec_all_le (vector unsigned char, vector unsigned char);
int vec_all_le (vector bool char, vector signed char);
int vec_all_le (vector signed char, vector bool char);
int vec_all_le (vector signed char, vector signed char);
int vec_all_le (vector bool short, vector unsigned short);
int vec_all_le (vector unsigned short, vector bool short);
int vec_all_le (vector unsigned short, vector unsigned short);
int vec_all_le (vector bool short, vector signed short);
int vec_all_le (vector signed short, vector bool short);
int vec_all_le (vector signed short, vector signed short);
int vec_all_le (vector bool int, vector unsigned int);
int vec_all_le (vector unsigned int, vector bool int);
int vec_all_le (vector unsigned int, vector unsigned int);
int vec_all_le (vector bool int, vector signed int);
int vec_all_le (vector signed int, vector bool int);
int vec_all_le (vector signed int, vector signed int);
int vec_all_le (vector float, vector float);

int vec_all_lt (vector bool char, vector unsigned char);
int vec_all_lt (vector unsigned char, vector bool char);
int vec_all_lt (vector unsigned char, vector unsigned char);
int vec_all_lt (vector bool char, vector signed char);
int vec_all_lt (vector signed char, vector bool char);
int vec_all_lt (vector signed char, vector signed char);
int vec_all_lt (vector bool short, vector unsigned short);
int vec_all_lt (vector unsigned short, vector bool short);
int vec_all_lt (vector unsigned short, vector unsigned short);
int vec_all_lt (vector bool short, vector signed short);
int vec_all_lt (vector signed short, vector bool short);
int vec_all_lt (vector signed short, vector signed short);
int vec_all_lt (vector bool int, vector unsigned int);
int vec_all_lt (vector unsigned int, vector bool int);
int vec_all_lt (vector unsigned int, vector unsigned int);
int vec_all_lt (vector bool int, vector signed int);
int vec_all_lt (vector signed int, vector bool int);
int vec_all_lt (vector signed int, vector signed int);
int vec_all_lt (vector float, vector float);

int vec_all_nan (vector float);

int vec_all_ne (vector signed char, vector bool char);
int vec_all_ne (vector signed char, vector signed char);
int vec_all_ne (vector unsigned char, vector bool char);
int vec_all_ne (vector unsigned char, vector unsigned char);
int vec_all_ne (vector bool char, vector bool char);
int vec_all_ne (vector bool char, vector unsigned char);
int vec_all_ne (vector bool char, vector signed char);
int vec_all_ne (vector signed short, vector bool short);
int vec_all_ne (vector signed short, vector signed short);
int vec_all_ne (vector unsigned short, vector bool short);
int vec_all_ne (vector unsigned short, vector unsigned short);
int vec_all_ne (vector bool short, vector bool short);
int vec_all_ne (vector bool short, vector unsigned short);
int vec_all_ne (vector bool short, vector signed short);
int vec_all_ne (vector pixel, vector pixel);
int vec_all_ne (vector signed int, vector bool int);
int vec_all_ne (vector signed int, vector signed int);
int vec_all_ne (vector unsigned int, vector bool int);
int vec_all_ne (vector unsigned int, vector unsigned int);
int vec_all_ne (vector bool int, vector bool int);
int vec_all_ne (vector bool int, vector unsigned int);
int vec_all_ne (vector bool int, vector signed int);
int vec_all_ne (vector float, vector float);

int vec_all_nge (vector float, vector float);

int vec_all_ngt (vector float, vector float);

int vec_all_nle (vector float, vector float);

int vec_all_nlt (vector float, vector float);

int vec_all_numeric (vector float);

int vec_any_eq (vector signed char, vector bool char);
int vec_any_eq (vector signed char, vector signed char);
int vec_any_eq (vector unsigned char, vector bool char);
int vec_any_eq (vector unsigned char, vector unsigned char);
int vec_any_eq (vector bool char, vector bool char);
int vec_any_eq (vector bool char, vector unsigned char);
int vec_any_eq (vector bool char, vector signed char);
int vec_any_eq (vector signed short, vector bool short);
int vec_any_eq (vector signed short, vector signed short);
int vec_any_eq (vector unsigned short, vector bool short);
int vec_any_eq (vector unsigned short, vector unsigned short);
int vec_any_eq (vector bool short, vector bool short);
int vec_any_eq (vector bool short, vector unsigned short);
int vec_any_eq (vector bool short, vector signed short);
int vec_any_eq (vector pixel, vector pixel);
int vec_any_eq (vector signed int, vector bool int);
int vec_any_eq (vector signed int, vector signed int);
int vec_any_eq (vector unsigned int, vector bool int);
int vec_any_eq (vector unsigned int, vector unsigned int);
int vec_any_eq (vector bool int, vector bool int);
int vec_any_eq (vector bool int, vector unsigned int);
int vec_any_eq (vector bool int, vector signed int);
int vec_any_eq (vector float, vector float);

int vec_any_ge (vector signed char, vector bool char);
int vec_any_ge (vector unsigned char, vector bool char);
int vec_any_ge (vector unsigned char, vector unsigned char);
int vec_any_ge (vector signed char, vector signed char);
int vec_any_ge (vector bool char, vector unsigned char);
int vec_any_ge (vector bool char, vector signed char);
int vec_any_ge (vector unsigned short, vector bool short);
int vec_any_ge (vector unsigned short, vector unsigned short);
int vec_any_ge (vector signed short, vector signed short);
int vec_any_ge (vector signed short, vector bool short);
int vec_any_ge (vector bool short, vector unsigned short);
int vec_any_ge (vector bool short, vector signed short);
int vec_any_ge (vector signed int, vector bool int);
int vec_any_ge (vector unsigned int, vector bool int);
int vec_any_ge (vector unsigned int, vector unsigned int);
int vec_any_ge (vector signed int, vector signed int);
int vec_any_ge (vector bool int, vector unsigned int);
int vec_any_ge (vector bool int, vector signed int);
int vec_any_ge (vector float, vector float);

int vec_any_gt (vector bool char, vector unsigned char);
int vec_any_gt (vector unsigned char, vector bool char);
int vec_any_gt (vector unsigned char, vector unsigned char);
int vec_any_gt (vector bool char, vector signed char);
int vec_any_gt (vector signed char, vector bool char);
int vec_any_gt (vector signed char, vector signed char);
int vec_any_gt (vector bool short, vector unsigned short);
int vec_any_gt (vector unsigned short, vector bool short);
int vec_any_gt (vector unsigned short, vector unsigned short);
int vec_any_gt (vector bool short, vector signed short);
int vec_any_gt (vector signed short, vector bool short);
int vec_any_gt (vector signed short, vector signed short);
int vec_any_gt (vector bool int, vector unsigned int);
int vec_any_gt (vector unsigned int, vector bool int);
int vec_any_gt (vector unsigned int, vector unsigned int);
int vec_any_gt (vector bool int, vector signed int);
int vec_any_gt (vector signed int, vector bool int);
int vec_any_gt (vector signed int, vector signed int);
int vec_any_gt (vector float, vector float);

int vec_any_le (vector bool char, vector unsigned char);
int vec_any_le (vector unsigned char, vector bool char);
int vec_any_le (vector unsigned char, vector unsigned char);
int vec_any_le (vector bool char, vector signed char);
int vec_any_le (vector signed char, vector bool char);
int vec_any_le (vector signed char, vector signed char);
int vec_any_le (vector bool short, vector unsigned short);
int vec_any_le (vector unsigned short, vector bool short);
int vec_any_le (vector unsigned short, vector unsigned short);
int vec_any_le (vector bool short, vector signed short);
int vec_any_le (vector signed short, vector bool short);
int vec_any_le (vector signed short, vector signed short);
int vec_any_le (vector bool int, vector unsigned int);
int vec_any_le (vector unsigned int, vector bool int);
int vec_any_le (vector unsigned int, vector unsigned int);
int vec_any_le (vector bool int, vector signed int);
int vec_any_le (vector signed int, vector bool int);
int vec_any_le (vector signed int, vector signed int);
int vec_any_le (vector float, vector float);

int vec_any_lt (vector bool char, vector unsigned char);
int vec_any_lt (vector unsigned char, vector bool char);
int vec_any_lt (vector unsigned char, vector unsigned char);
int vec_any_lt (vector bool char, vector signed char);
int vec_any_lt (vector signed char, vector bool char);
int vec_any_lt (vector signed char, vector signed char);
int vec_any_lt (vector bool short, vector unsigned short);
int vec_any_lt (vector unsigned short, vector bool short);
int vec_any_lt (vector unsigned short, vector unsigned short);
int vec_any_lt (vector bool short, vector signed short);
int vec_any_lt (vector signed short, vector bool short);
int vec_any_lt (vector signed short, vector signed short);
int vec_any_lt (vector bool int, vector unsigned int);
int vec_any_lt (vector unsigned int, vector bool int);
int vec_any_lt (vector unsigned int, vector unsigned int);
int vec_any_lt (vector bool int, vector signed int);
int vec_any_lt (vector signed int, vector bool int);
int vec_any_lt (vector signed int, vector signed int);
int vec_any_lt (vector float, vector float);

int vec_any_nan (vector float);

int vec_any_ne (vector signed char, vector bool char);
int vec_any_ne (vector signed char, vector signed char);
int vec_any_ne (vector unsigned char, vector bool char);
int vec_any_ne (vector unsigned char, vector unsigned char);
int vec_any_ne (vector bool char, vector bool char);
int vec_any_ne (vector bool char, vector unsigned char);
int vec_any_ne (vector bool char, vector signed char);
int vec_any_ne (vector signed short, vector bool short);
int vec_any_ne (vector signed short, vector signed short);
int vec_any_ne (vector unsigned short, vector bool short);
int vec_any_ne (vector unsigned short, vector unsigned short);
int vec_any_ne (vector bool short, vector bool short);
int vec_any_ne (vector bool short, vector unsigned short);
int vec_any_ne (vector bool short, vector signed short);
int vec_any_ne (vector pixel, vector pixel);
int vec_any_ne (vector signed int, vector bool int);
int vec_any_ne (vector signed int, vector signed int);
int vec_any_ne (vector unsigned int, vector bool int);
int vec_any_ne (vector unsigned int, vector unsigned int);
int vec_any_ne (vector bool int, vector bool int);
int vec_any_ne (vector bool int, vector unsigned int);
int vec_any_ne (vector bool int, vector signed int);
int vec_any_ne (vector float, vector float);

int vec_any_nge (vector float, vector float);

int vec_any_ngt (vector float, vector float);

int vec_any_nle (vector float, vector float);

int vec_any_nlt (vector float, vector float);

int vec_any_numeric (vector float);

int vec_any_out (vector float, vector float);
```

6.47. 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; Section 6.25 *Declaring Attributes of Functions*, for further explanation.

6.47.1. RS/6000 and PowerPC Pragmas

The RS/6000 and PowerPC targets define one pragma for controlling whether or not the `longcall` attribute is added to function declarations by default. This pragma overrides the `-mlongcall` option, but not the `longcall` and `shortcall` attributes. Section 4.17.1 *IBM RS/6000 and PowerPC Options*, for more information about when long calls are and are not necessary.

`longcall (1)`

Apply the `longcall` attribute to all subsequent function declarations.

`longcall (0)`

Do not apply the `longcall` attribute to subsequent function declarations.

6.47.2. Darwin Pragmas

The following pragmas are available for all architectures running the Darwin operating system. These are useful for compatibility with other Mac OS 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.

6.48. 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.

6.49. Thread-Local Storage

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. The run-time model GCC uses to implement this originates in the IA-64 processor-specific ABI, but has since been migrated to other processors as well. It requires significant support from the linker (`ld`), dynamic linker (`ld.so`), and system libraries (`libc.so` and `libpthread.so`), so it is not available everywhere.

At the user level, the extension is visible with a new storage class keyword: `__thread`. For example:

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

The `__thread` specifier may be used alone, with the `extern` or `static` specifiers, but with no other storage class specifier. When used with `extern` or `static`, `__thread` must appear immediately after the other storage class specifier.

The `__thread` specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data member.

When the address-of operator is applied to a thread-local variable, it is evaluated at run-time and returns the address of the current thread's instance of that variable. An address so obtained may be used by any thread. When a thread terminates, any pointers to thread-local variables in that thread become invalid.

No static initialization may refer to the address of a thread-local variable.

In C++, if an initializer is present for a thread-local variable, it must be a `constant-expression`, as defined in 5.19.2 of the ANSI/ISO C++ standard.

See http://people.redhat.com/drepper/tls.pdfELF Handling For Thread-Local Storage for a detailed explanation of the four thread-local storage addressing models, and how the run-time is expected to function.

6.49.1. ISO/IEC 9899:1999 Edits for Thread-Local Storage

The following are a set of changes to ISO/IEC 9899:1999 (aka C99) that document the exact semantics of the language extension.

- [5.1.2 Execution environments]

  Add new text after paragraph 1

  > Within either execution environment, a *thread* is a flow of control within a program. It is implementation defined whether or not there may be more than one thread associated with a program. It is implementation defined how threads beyond the first are created, the name and type of the function called at thread startup, and how threads may be terminated. However, objects with thread storage duration shall be initialized before thread startup.

- [6.2.4 Storage durations of objects]

  Add new text before paragraph 3

  > An object whose identifier is declared with the storage-class specifier `__thread` has *thread storage duration*. Its lifetime is the entire execution of the thread, and its stored value is initialized only once, prior to thread startup.

- [6.4.1 Keywords]

  Add `__thread`.

- [6.7.1 Storage-class specifiers]

  Add `__thread` to the list of storage class specifiers in paragraph 1.

  Change paragraph 2 to

  > With the exception of `__thread`, at most one storage-class specifier may be given [. . .]. The `__thread` specifier may be used alone, or immediately following `extern` or `static`.

  Add new text after paragraph 6

  > The declaration of an identifier for a variable that has block scope that specifies `__thread` shall also specify either `extern` or `static`.
  >
  > The `__thread` specifier shall be used only with variables.

6.49.2. ISO/IEC 14882:1998 Edits for Thread-Local Storage

The following are a set of changes to ISO/IEC 14882:1998 (aka C++98) that document the exact semantics of the language extension.

- *[intro.execution]*

  New text after paragraph 4

  > A *thread* is a flow of control within the abstract machine. It is implementation defined whether or not there may be more than one thread.

  New text after paragraph 7

  > It is unspecified whether additional action must be taken to ensure when and whether side effects are visible to other threads.

- *[lex.key]*

  Add `__thread`.

- *[basic.start.main]*

  Add after paragraph 5

  > The thread that begins execution at the `main` function is called the *main thread*. It is implementation defined how functions beginning threads other than the main thread are designated or typed. A function so designated, as well as the `main` function, is called a *thread startup function*. It is implementation defined what happens if a thread startup function returns. It is implementation defined what happens to other threads when any thread calls `exit`.

- *[basic.start.init]*

  Add after paragraph 4

  > The storage for an object of thread storage duration shall be statically initialized before the first statement of the thread startup function. An object of thread storage duration shall not require dynamic initialization.

- *[basic.start.term]*

  Add after paragraph 3

  > The type of an object with thread storage duration shall not have a non-trivial destructor, nor shall it be an array type whose elements (directly or indirectly) have non-trivial destructors.

- *[basic.stc]*

  Add "thread storage duration" to the list in paragraph 1.

  Change paragraph 2

  > Thread, static, and automatic storage durations are associated with objects introduced by declarations [. . .].

  Add `__thread` to the list of specifiers in paragraph 3.

- *[basic.stc.thread]*

  New section before *[basic.stc.static]*

  > The keyword `__thread` applied to a non-local object gives the object thread storage duration.
  >
  > A local variable or class data member declared both `static` and `__thread` gives the variable or member thread storage duration.

- *[basic.stc.static]*

  Change paragraph 1

  > All objects which have neither thread storage duration, dynamic storage duration nor are local [. . .].

- *[dcl.stc]*

  Add `__thread` to the list in paragraph 1.

  Change paragraph 1

  > With the exception of `__thread`, at most one `storage-class-specifier` shall appear in a given `decl-specifier-seq`. The `__thread` specifier may be used alone, or immediately following the `extern` or `static` specifiers. [. . .]

  Add after paragraph 5

  > The `__thread` specifier can be applied only to the names of objects and to anonymous unions.

- *[class.mem]*

  Add after paragraph 6

  > Non-`static` members shall not be `__thread`.

Chapter 7. 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++ ().

7.1. 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. The GNU C `typeof` extension allows you to write safe macros that avoid this kind of problem (Section 6.6 *Referring to a Type with `typeof`*). 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.

7.2. 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.

7.3. Restricting Pointer Aliasing

As with the C front end, 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.

7.4. 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. Section 7.6 *Where's the Template?*.

When used with GNU ld version 2.8 or later on an ELF system such as GNU/Linux, 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.

Section 7.5 *#pragma interface and implementation*, for another way to control placement of these constructs.

7.5. #pragma interface and implementation

`#pragma interface` and `#pragma implementation` provide the user with a way of explicitly directing the compiler to emit entities with vague linkage (and debugging information) in a particular translation unit.

Note: As of GCC 2.7.2, these `#pragma`s are not useful in most cases, because of COMDAT support and the "key method" heuristic mentioned in Section 7.4 *Vague Linkage*. Using them can actually cause your program to grow due to unnecesary out-of-line copies of inline functions. Currently the only benefit of these `#pragma`s is reduced duplication of debugging information, and that should be addressed soon on DWARF 2 targets with the use of COMDAT sections.

```
#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.

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 an inline function defined in that class is similar to an explicit `extern` declaration--the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as `#pragma implementation`, the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with `-fno-implement-inlines`. If any calls were not inlined, you will get linker errors.

7.6. 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 are referred 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

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`.

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 GNU/Linux, 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 given in the ISO standard 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.

7.7. 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.

7.8. 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.

See also Section 7.9 *Strong Using*.

7.9. Strong Using

Caution: The semantics of this extension are not fully defined. Users should refrain from using this extension as its semantics may change subtly over time. It is possible that this extension wil be removed in future versions of G++.

A using-directive with `__attribute ((strong))` is stronger than a normal using-directive in two ways:

- Templates from the used namespace can be specialized as though they were members of the using namespace.
- The using namespace is considered an associated namespace of all templates in the used namespace for purposes of argument-dependent name lookup.

This is useful for composing a namespace transparently from implementation namespaces. For example:

```
namespace std {
  namespace debug {
    template <class T> struct A { };
  }
  using namespace debug __attribute ((__strong__));
  template <> struct A<int> { };   // ok to specialize

  template <class T> void f (A<T>);
}

int main()
{
  f (std::A<float>());             // lookup finds std::f
  f (std::A<int>());
}
```

7.10. Offsetof

G++ uses a syntactic extension to implement the `offsetof` macro.

In particular:

```
__offsetof__ (expression)
```

is equivalent to the parenthesized expression, except that the expression is considered an integral constant expression even if it contains certain operators that are not normally permitted in an integral constant expression. Users should never use `__offsetof__` directly; the only valid use of `__offsetof__` is to implement the `offsetof` macro in <`stddef.h`>.

7.11. 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.

7.12. 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. Section 7.6 *Where's the Template?*. 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 Section 7.13 *Backwards Compatibility*.

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 is now removed from G++.

The use of default arguments in function pointers, function typedefs and and other places where they are not permitted by the standard is deprecated and will be removed from a future version of G++.

7.13. 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 Section 7.12 *Deprecated Features*.

`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.

Chapter 8. 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 mailto:ovidiu@@cup.hp.com.

8.1. `+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`.

8.1.1. 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.

8.2. 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}`

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.

8.3. 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.

8.4. 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 by default 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 MyConstantStringClass
{
  Class isa;
  char *c_string;
  unsigned int len;
}
@end
```

NXConstantString inherits from Object; 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, but the final ivar layout of the class must be the compatible with the given structure.

When the compiler creates the statically allocated constant string object, the c_string field will be filled by the compiler with the string; the length field will be filled by the compiler with the string length; the isa pointer will be filled with NULL by the compiler, and it will later be fixed up automatically at runtime by the GNU Objective-C runtime library to point to the class which was set by the -fconstant-string-class option when the object file is loaded (if you wonder how it works behind the scenes, the name of the class to use, and the list of static objects to fixup, are stored by the compiler in the object file in a place where the GNU runtime library will find them at runtime).

As a result, 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.

8.5. 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.

Chapter 9. Binary Compatibility

Binary compatibility encompasses several related concepts:

application binary interface (ABI)

The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.

ABI conformance

A compiler conforms to an ABI if it generates code that follows all of the specifications enumerated by that ABI. A library conforms to an ABI if it is implemented according to that ABI. An application conforms to an ABI if it is built using tools that conform to that ABI and does not contain source code that specifically changes behavior specified by the ABI.

calling conventions

Calling conventions are a subset of an ABI that specify of how arguments are passed and function results are returned.

interoperability

Different sets of tools are interoperable if they generate files that can be used in the same program. The set of tools includes compilers, assemblers, linkers, libraries, header files, startup files, and debuggers. Binaries produced by different sets of tools are not interoperable unless they implement the same ABI. This applies to different versions of the same tools as well as tools from different vendors.

intercallability

Whether a function in a binary built by one set of tools can call a function in a binary built by a different set of tools is a subset of interoperability.

implementation-defined features

Language standards include lists of implementation-defined features whose behavior can vary from one implementation to another. Some of these features are normally covered by a platform's ABI and others are not. The features that are not covered by an ABI generally affect how a program behaves, but not intercallability.

compatibility

Conformance to the same ABI and the same behavior of implementation-defined features are both relevant for compatibility.

The application binary interface implemented by a C or C++ compiler affects code generation and runtime support for:

- size and alignment of data types
- layout of structured types
- calling conventions
- register usage conventions

- interfaces for runtime arithmetic support
- object file formats

In addition, the application binary interface implemented by a C++ compiler affects code generation and runtime support for:

- name mangling
- exception handling
- invoking constructors and destructors
- layout, alignment, and padding of classes
- layout and alignment of virtual tables

Some GCC compilation options cause the compiler to generate code that does not conform to the platform's default ABI. Other options cause different program behavior for implementation-defined features that are not covered by an ABI. These options are provided for consistency with other compilers that do not follow the platform's default ABI or the usual behavior of implementation-defined features for the platform. Be very careful about using such options.

Most platforms have a well-defined ABI that covers C code, but ABIs that cover C++ functionality are not yet common.

Starting with GCC 3.2, GCC binary conventions for C++ are based on a written, vendor-neutral C++ ABI that was designed to be specific to 64-bit Itanium but also includes generic specifications that apply to any platform. This C++ ABI is also implemented by other compiler vendors on some platforms, notably GNU/Linux and BSD systems. We have tried hard to provide a stable ABI that will be compatible with future GCC releases, but it is possible that we will encounter problems that make this difficult. Such problems could include different interpretations of the C++ ABI by different vendors, bugs in the ABI, or bugs in the implementation of the ABI in different compilers. GCC's `-Wabi` switch warns when G++ generates code that is probably not compatible with the C++ ABI.

The C++ library used with a C++ compiler includes the Standard C++ Library, with functionality defined in the C++ Standard, plus language runtime support. The runtime support is included in a C++ ABI, but there is no formal ABI for the Standard C++ Library. Two implementations of that library are interoperable if one follows the de-facto ABI of the other and if they are both built with the same compiler, or with compilers that conform to the same ABI for C++ compiler and runtime support.

When G++ and another C++ compiler conform to the same C++ ABI, but the implementations of the Standard C++ Library that they normally use do not follow the same ABI for the Standard C++ Library, object files built with those compilers can be used in the same program only if they use the same C++ library. This requires specifying the location of the C++ library header files when invoking the compiler whose usual library is not being used. The location of GCC's C++ header files depends on how the GCC build was configured, but can be seen by using the G++ `-v` option. With default configuration options for G++ 3.3 the compile line for a different C++ compiler needs to include

```
    -Igcc_install_directory/include/c++/3.3
```

Similarly, compiling code with G++ that must use a C++ library other than the GNU C++ library requires specifying the location of the header files for that other library.

The most straightforward way to link a program to use a particular C++ library is to use a C++ driver that specifies that C++ library by default. The `g++` driver, for example, tells the linker where to find GCC's C++ library (`libstdc++`) plus the other libraries and startup files it needs, in the proper order.

If a program must use a different C++ library and it's not possible to do the final link using a C++ driver that uses that library by default, it is necessary to tell `g++` the location and name of that library. It might also be necessary to specify different startup files and other runtime support libraries, and to suppress the use of GCC's support libraries with one or more of the options `-nostdlib`, `-nostartfiles`, and `-nodefaultlibs`.

Chapter 10. gcov--a Test Coverage Program

gcov is a tool you can use in conjunction with GCC to test code coverage in your programs.

10.1. 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 and to discover untested parts of your program. 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.

10.2. 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.

`-a`
`-all-blocks`

Write individual execution counts for every basic block. Normally gcov outputs execution counts only for the main blocks of a line. With this option you can determine if blocks within a single line are not being executed.

`-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. Unconditional branches will not be shown, unless the `-u` option is given.

`-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. If you uses the `-p` option, both the including and included file names will be complete path names.

`-p`
`-preserve-paths`

Preserve complete path information in the names of generated `.gcov` files. Without this option, just the filename component is used. With this option, all directories are used, with '/' characters translated to '#' characters, '.' directory components removed and '..' components renamed to '^'. This is useful if sourcefiles are in several different directories. It also affects the `-l` option.

`-f`
`-function-summaries`

Output summaries for each function in addition to the file level summary.

`-o directory|file`
`-object-directory directory`
`-object-file file`

Specify either the directory containing the gcov data files, or the object path name. The `.gcno`, and `.gcda` data files are searched for using this option. If a directory is specified, the data files are in that directory and named after the source file name, without its extension. If a file is specified here, the data files are named after that file, without its extension. If this option is not supplied, it defaults to the current directory.

`-u`
`-unconditional-branches`

When branch counts are given, include those of unconditional branches. Unconditional branches are normally not interesting.

`gcov` should be run with the current directory the same as that when you invoked the compiler. Otherwise it will not be able to locate the source files. `gcov` produces files called `mangledname.gcov` in the current directory. These contain the coverage information of the source file they correspond to. One `.gcov` file is produced for each source file containing code, which was compiled to produce the data files. The `mangledname` part of the output file name is usually simply the source file name, but can be something more complicated if the `-l` or `-p` options are given. Refer to those options for details.

The `.gcov` files contain the ':' separated fields along with program source code. The format is

```
execution_count:line_number:source line text
```

Additional block information may succeed each line, when requested by command line option. The `execution_count` is `-` for lines containing no code and `#####` for lines which were never executed. Some lines of information at the start have `line_number` of zero.

When printing percentages, 0% and 100% are only printed when the values are *exactly* 0% and 100% respectively. Other values which would conventionally be rounded to 0% or 100% are instead printed as the nearest non-boundary value.

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 object file is located.

Running the program will cause profile output to be generated. For each source file compiled with `-fprofile-arcs`, an accompanying `.gcda` file will be placed in the object file 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
90.00% of 10 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

The file `tmp.c.gcov` contains output from `gcov`. Here is a sample:

```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
    #####:   13:    printf ("Failure\n");
        -:   14:  else
        1:   15:    printf ("Success\n");
        1:   16:  return 0;
        -:   17:}
```

When you use the -a option, you will get individual block counts, and the output looks like this:

```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
        1:    4:{
        1:    4-block  0
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
       11:    9-block  0
       10:   10:    total += i;
       10:   10-block  0
        -:   11:
        1:   12:  if (total != 45)
        1:   12-block  0
    #####:   13:    printf ("Failure\n");
    $$$$$:   13-block  0
        -:   14:  else
        1:   15:    printf ("Success\n");
        1:   15-block  0
        1:   16:  return 0;
        1:   16-block  0
        -:   17:}
```

In this mode, each basic block is only shown on one line - the last line of the block. A multi-line block will only contribute to the execution count of that last line, and other lines will not be shown to contain code, unless previous blocks end on those lines. The total execution count of a line is shown and subsequent lines show the execution counts for individual blocks that end on that line. After each block, the branch and call counts of the block will be shown, if the `-b` option is given.

Because of the way GCC instruments calls, a call count can be shown after a line with no individual blocks. As you can see, line 13 contains a basic block that was not executed.

When you use the `-b` option, your output looks like this:

```
$ gcov -b tmp.c
90.00% of 10 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:

```
        -:    0:Source:tmp.c
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:int main (void)
function main called 1 returned 1 blocks executed 75%
        1:    4:{
        1:    5:  int i, total;
        -:    6:
        1:    7:  total = 0;
        -:    8:
       11:    9:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:   10:    total += i;
        -:   11:
        1:   12:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   13:    printf ("Failure\n");
call    0 never executed
        -:   14:  else
        1:   15:    printf ("Success\n");
call    0 called 1 returned 100%
        1:   16:  return 0;
        -:   17:}
```

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 `.gcda` 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 `.gcda` 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 `.gcda` 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.

10.3. 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:   12:if (a != b)
      100:   13:  c = 1;
      100:   14:else
      100:   15:  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.

Inlineable functions can create unexpected line counts. Line counts are shown for the source code of the inlineable function, but what is shown depends on where the function is inlined, or if it is not inlined at all.

If the function is not inlined, the compiler must emit an out of line copy of the function, in any object file that needs it. If `fileA.o` and `fileB.o` both contain out of line bodies of a particular inlineable function, they will also both contain coverage counts for that function. When `fileA.o` and `fileB.o` are linked together, the linker will, on many systems, select one of those out of line bodies for all calls to that function, and remove or ignore the other. Unfortunately, it will not remove the coverage counters for the unused function body. Hence when instrumented, all but one use of that function will show zero counts.

If the function is inlined in several places, the block structure in each location might not be the same. For instance, a condition might now be calculable at compile time in some instances. Because the coverage of all the uses of the inline function will be shown for the same source lines, the line counts themselves might seem inconsistent.

10.4. Brief description of `gcov`data files

`gcov` uses two files for profiling. The names of these files are derived from the original *object* file by substituting the file suffix with either `.gcno`, or `.gcda`. All of these files are placed in the same directory as the object file, and contain data stored in a platform-independent format.

The `.gcno` file is generated when the source file is compiled with the GCC `-ftest-coverage` option. It contains information to reconstruct the basic block graphs and assign source line numbers to blocks.

The `.gcda` file is generated when a program containing object files built with the GCC `-fprofile-arcs` option is executed. A separate `.gcda` file is created for each object file compiled with this option. It contains arc transition counts, and some summary information.

The full details of the file format is specified in `gcov-io.h`, and functions provided in that header file should be used to access the coverage files.

Chapter 11. 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.

11.1. 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.

11.2. 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. .
- At present, the program `mips-tfile` which adds debug support to object files on MIPS systems does not work in a cross compile environment.

11.3. 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.

- On many platforms, GCC supports a different ABI for C++ than do other compilers, so the object files compiled by GCC cannot be used with object files generated by another C++ compiler.

An area where the difference is most apparent is name mangling. The use of different name mangling 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 that can be done about these problems.
- 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.

- 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.
- 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.
- 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 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`.

11.4. 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, 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
  ```

11.5. Incompatibilities of GCC

There are several noteworthy incompatibilities between GNU C and K&R (non-ISO) versions of C.

- 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.

- `-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.

- 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.

- 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.

- K&R compilers allow comments to cross over an inclusion boundary (i.e. started in an include file and ended in the including file).

- 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.

- 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`.

- PCC allows typedef names to be used as function parameters.

- Traditional C 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. 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.

- 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 `/*...*/`.

- 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 target hook `TARGET_STRUCT_VALUE_RTX` tells 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.

11.6. 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.

11.7. Standard Libraries

GCC by itself attempts to be a conforming freestanding implementation. Chapter 3 *Language Standards Supported by GCC*, 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.

11.8. 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 (Section 4.10 *Options That Control Optimization*).

- 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.

- On AIX, GCC scans object files and library archives for static constructors and destructors when linking an application before the linker prunes unreferenced symbols. This is necessary to prevent the AIX linker from mistakenly assuming that static constructor or destructor are unused and removing them before the scanning can occur. All static constructors and destructors found will be referenced even though the modules in which they occur may not be used by the program. This may lead to both increased executable size and unexpected symbol references.

11.9. 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.

11.9.1. 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.

11.9.2. Name lookup, templates, and accessing members of base classes

The C++ standard prescribes that all names that are not dependent on template parameters are bound to their present definitions when parsing a template function or class.[1] Only names that are dependent are looked up at the point of instantiation. For example, consider

```
  void foo(double);

  struct A {
    template <typename T>
    void f () {
      foo (1);        // 1
      int i = N;      // 2
      T t;
      t.bar();        // 3
      foo (t);        // 4
    }

    static const int N;
  };
```

1. The C++ standard just uses the term "dependent" for names that depend on the type or value of template parameters. This shorter term will also be used in the rest of this section.

Here, the names foo and N appear in a context that does not depend on the type of T. The compiler will thus require that they are defined in the context of use in the template, not only before the point of instantiation, and will here use ::foo(double) and A::N, respectively. In particular, it will convert the integer value to a double when passing it to ::foo(double).

Conversely, bar and the call to foo in the fourth marked line are used in contexts that do depend on the type of T, so they are only looked up at the point of instantiation, and you can provide declarations for them after declaring the template, but before instantiating it. In particular, if you instantiate A::f<int>, the last line will call an overloaded ::foo(int) if one was provided, even if after the declaration of struct A.

This distinction between lookup of dependent and non-dependent names is called two-stage (or dependent) name lookup. G++ implements it since version 3.4.

Two-stage name lookup sometimes leads to situations with behavior different from non-template codes. The most common is probably this:

```
template <typename T> struct Base {
  int i;
};

template <typename T> struct Derived : public Base<T> {
  int get_i() { return i; }
};
```

In get_i(), i is not used in a dependent context, so the compiler will look for a name declared at the enclosing namespace scope (which is the global scope here). It will not look into the base class, since that is dependent and you may declare specializations of Base even after declaring Derived, so the compiler can't really know what i would refer to. If there is no global variable i, then you will get an error message.

In order to make it clear that you want the member of the base class, you need to defer lookup until instantiation time, at which the base class is known. For this, you need to access i in a dependent context, by either using this->i (remember that this is of type Derived<T>*, so is obviously dependent), or using Base<T>::i. Alternatively, Base<T>::i might be brought into scope by a using-declaration.

Another, similar example involves calling member functions of a base class:

```
template <typename T> struct Base {
    int f();
};

template <typename T> struct Derived : Base<T> {
    int g() { return f(); };
};
```

Again, the call to f() is not dependent on template arguments (there are no arguments that depend on the type T, and it is also not otherwise specified that the call should be in a dependent context). Thus a global declaration of such a function must be available, since the one in the base class is not visible until instantiation time. The compiler will consequently produce the following error message:

```
x.cc: In member function 'int Derived<T>::g()':
x.cc:6: error: there are no arguments to 'f' that depend on a template
   parameter, so a declaration of 'f' must be available
```

```
  x.cc:6: error: (if you use '-fpermissive', G++ will accept your code, but
     allowing the use of an undeclared name is deprecated)
```

To make the code valid either use `this->f()`, or `Base<T>::f()`. Using the `-fpermissive` flag will also let the compiler accept the code, by marking all function calls for which no declaration is visible at the time of definition of the template for later lookup at instantiation time, as if it were a dependent call. We do not recommend using `-fpermissive` to work around invalid code, and it will also only catch cases where functions in base classes are called, not where variables in base classes are used (as in the example above).

Note that some compilers (including G++ versions prior to 3.4) get these examples wrong and accept above code without an error. Those compilers do not implement two-stage name lookup correctly.

11.9.3. 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:

```
const string& tmp = strfunc ();
charfunc (tmp.c_str ());
```

11.9.4. 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 more than once. 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.

11.10. 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.)

11.11. 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.

  C contains many standard functions that return a value that most programs choose to ignore. One obvious example is `printf`. Warning about this practice only leads the defensive programmer to clutter programs with dozens of casts to `void`. Such casts are required so frequently that they become visual noise. Writing those casts becomes so automatic that they no longer convey useful information about the intentions of the programmer. For functions where the return value should never be ignored, use the `warn_unused_result` function attribute (Section 6.25 *Declaring Attributes of Functions*).

- 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__` unconditionally. 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 testsuites call this "failure", they should be run with the GCC option `-pedantic-errors`, which will turn these warnings into errors.

11.12. 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.

Section 4.8 *Options to Request or Suppress Warnings*, for more detail on these and related command-line options.

Chapter 12. 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.

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 `-foption` 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.

Keyword Index