Using the GNU Compiler Collection

Richard M. Stallman

Last updated 28 July 1999

for gcc-2.95

(DOC-0464-00)

For GCC Version 2.95

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA
Last printed April, 1998.
Printed copies are available for $50 each.
ISBN 1-882114-37-X

1 Compile C, C++, Objective C, or Fortran

The C, C++, and Objective C, and Fortran versions of the compiler are integrated; this is why we use the name "GNU Compiler Collection". GCC can compile programs written in C, C++, Objective C, or Fortran. The Fortran compiler is described in a separate manual.

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

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

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

Front ends for other languages, such as Ada 9X, Fortran, Modula-3, and Pascal, are under development. These front-ends, like that for C++, are built in subdirectories of GCC and link to it. The result is an integrated compiler that can compile programs written in C, C++, Objective C, or any of the languages for which you have installed front ends.

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

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

2 GCC Command Options

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

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

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

See Section 2.3 [Compiling C++ Programs], page 11, for a summary of special options for compiling C++ programs.

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

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

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

2.1 Option Summary

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

Overall Options

See Section 2.2 [Options Controlling the Kind of Output], page 10.

```
-c  -S  -E  -o file  -pipe  -v  --help  -x language
```

C Language Options

See Section 2.4 [Options Controlling C Dialect], page 12.

```
-ansi -fstd  -fallow-single-precision  -fcond-mismatch  -fno-asm
-fno-builtin  -ffreestanding  -fhosted  -fsigned-bitfields  -fsigned-char
-funsigned-bitfields  -funsigned-char  -fwritable-strings
-traditional  -traditional-cpp  -trigraphs
```

C++ Language Options

See Section 2.5 [Options Controlling C++ Dialect], page 16.

```
-fno-access-control  -fcheck-new  -fconserve-space  -fdollars-in-identifiers
-fno-elide-constructors  -fexternal-templates  -ffor-scope
-fno-for-scope  -fno-gnu-keywords  -fguiding-decls  -fhandle-signatures
```

```
-fhonor-std -fhuge-objects  -fno-implicit-templates  -finit-priority
-fno-implement-inlines -fname-mangling-version-n  -fno-default-inline
-foperator-names  -fno-optional-diags  -fpermissive -frepo  -fstrict-prototy
-fsquangle  -ftemplate-depth-n  -fthis-is-variable  -fvtable-thunks
-nostdinc++  -Wctor-dtor-privacy -Wno-deprecated -Weffc++
-Wno-non-template-friend
-Wnon-virtual-dtor  -Wold-style-cast  -Woverloaded-virtual
-Wno-pmf-conversions  -Wreorder  -Wsign-promo  -Wsynth
```

Warning Options

See Section 2.6 [Options to Request or Suppress Warnings], page 22.

```
-fsyntax-only  -pedantic  -pedantic-errors
-w  -W  -Wall  -Waggregate-return  -Wbad-function-cast
-Wcast-align  -Wcast-qual  -Wchar-subscripts  -Wcomment
-Wconversion  -Werror  -Wformat
-Wid-clash-len  -Wimplicit -Wimplicit-int
-Wimplicit-function-declaration  -Wimport
-Werror-implicit-function-declaration  -Winline
-Wlarger-than-len  -Wlong-long
-Wmain  -Wmissing-declarations  -Wmissing-noreturn
-Wmissing-prototypes  -Wmultichar  -Wnested-externs  -Wno-import
-Wparentheses -Wpointer-arith  -Wredundant-decls
-Wreturn-type -Wshadow  -Wsign-compare  -Wstrict-prototypes
-Wswitch  -Wtraditional
-Wtrigraphs -Wundef  -Wuninitialized  -Wunused  -Wwrite-strings
-Wunknown-pragmas
```

Debugging Options

See Section 2.7 [Options for Debugging Your Program or GCC], page 29.

```
-a  -ax  -dletters  -fdump-unnumbered -fpretend-float
-fprofile-arcs  -ftest-coverage
-g  -glevel  -gcoff  -gdwarf  -gdwarf-1  -gdwarf-1+  -gdwarf-2
-ggdb  -gstabs  -gstabs+  -gxcoff  -gxcoff+
-p  -pg  -print-file-name=library  -print-libgcc-file-name
-print-prog-name=program  -print-search-dirs  -save-temps
```

Optimization Options

See Section 2.8 [Options that Control Optimization], page 35.

```
-fbranch-probabilities  -foptimize-register-moves
-fcaller-saves  -fcse-follow-jumps  -fcse-skip-blocks
-fdelayed-branch   -fexpensive-optimizations
-ffast-math  -ffloat-store  -fforce-addr  -fforce-mem
-fdata-sections -ffunction-sections  -fgcse
-finline-functions -finline-limit-n -fkeep-inline-functions
-fno-default-inline -fno-defer-pop  -fno-function-cse
-fno-inline  -fno-peephole  -fomit-frame-pointer -fregmove
-frerun-cse-after-loop  -frerun-loop-opt -fschedule-insns
-fschedule-insns2  -fstrength-reduce  -fthread-jumps
-funroll-all-loops  -funroll-loops
-fmove-all-movables  -freduce-all-givs -fstrict-aliasing
```

```
-O  -O0  -O1  -O2  -O3 -Os
```

Preprocessor Options
: See Section 2.9 [Options Controlling the Preprocessor], page 41.

```
-Aquestion(answer)  -C  -dD  -dM  -dN
-Dmacro[=defn]  -E  -H
-idirafter dir
-include file  -imacros file
-iprefix file  -iwithprefix dir
-iwithprefixbefore dir  -isystem dir -isystem-c++ dir
-M  -MD  -MM  -MMD  -MG  -nostdinc  -P  -trigraphs
-undef  -Umacro  -Wp,option
```

Assembler Option
: See Section 2.10 [Passing Options to the Assembler], page 44.

```
-Wa,option
```

Linker Options
: See Section 2.11 [Options for Linking], page 44.

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

Directory Options
: See Section 2.12 [Options for Directory Search], page 46.

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

Target Options
: See Section 2.13 [Target Options], page 47.

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

Machine Dependent Options
: See Section 2.14 [Hardware Models and Configurations], page 48.

M680x0 Options

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

VAX Options

```
-mg  -mgnu  -munix
```

SPARC Options

```
-mcpu=cpu type
-mtune=cpu type
-mcmodel=code model
-malign-jumps=num  -malign-loops=num
-malign-functions=num
-m32  -m64
```

```
-mapp-regs  -mbroken-saverestore  -mcypress  -mepilogue
-mflat  -mfpu  -mhard-float  -mhard-quad-float
-mimpure-text  -mlive-g0  -mno-app-regs  -mno-epilogue
-mno-flat  -mno-fpu  -mno-impure-text
-mno-stack-bias  -mno-unaligned-doubles
-msoft-float  -msoft-quad-float  -msparclite  -mstack-bias
-msupersparc  -munaligned-doubles  -mv8
```

Convex Options

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

AMD29K Options

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

ARM Options

```
-mapcs-frame -mno-apcs-frame
-mapcs-26 -mapcs-32
-mapcs-stack-check -mno-apcs-stack-check
-mapcs-float -mno-apcs-float
-mapcs-reentrant -mno-apcs-reentrant
-msched-prolog -mno-sched-prolog
-mlittle-endian -mbig-endian -mwords-little-endian
-mshort-load-bytes -mno-short-load-bytes -mshort-load-words -mno-short-load-
-msoft-float -mhard-float -mfpe
-mthumb-interwork -mno-thumb-interwork
-mcpu= -march= -mfpe=
-mstructure-size-boundary=
-mbsd -mxopen -mno-symrename
-mabort-on-noreturn
-mno-sched-prolog
```

Thumb Options

```
-mtpcs-frame -mno-tpcs-frame
-mtpcs-leaf-frame -mno-tpcs-leaf-frame
-mlittle-endian  -mbig-endian
-mthumb-interwork -mno-thumb-interwork
-mstructure-size-boundary=
```

MN10200 Options

```
-mrelax
```

MN10300 Options
```
-mmult-bug
-mno-mult-bug
-mrelax
```

M32R/D Options
-mcode-model=*model type*   -msdata=*sdata type*
-G *num*

M88K Options
```
-m88000  -m88100  -m88110  -mbig-pic
-mcheck-zero-division  -mhandle-large-shift
-midentify-revision  -mno-check-zero-division
-mno-ocs-debug-info  -mno-ocs-frame-position
-mno-optimize-arg-area  -mno-serialize-volatile
-mno-underscores  -mocs-debug-info
-mocs-frame-position  -moptimize-arg-area
```
-mserialize-volatile -mshort-data-*num* -msvr3
```
-msvr4  -mtrap-large-shift  -muse-div-instruction
-mversion-03.00  -mwarn-passed-structs
```

RS/6000 and PowerPC Options
-mcpu=*cpu type*
-mtune=*cpu type*
```
-mpower  -mno-power  -mpower2  -mno-power2
-mpowerpc  -mno-powerpc
-mpowerpc-gpopt  -mno-powerpc-gpopt
-mpowerpc-gfxopt  -mno-powerpc-gfxopt
-mnew-mnemonics  -mno-new-mnemonics
-mfull-toc   -mminimal-toc  -mno-fop-in-toc  -mno-sum-in-toc
-maix64  -maix32  -mxl-call  -mno-xl-call  -mthreads  -mpe
-msoft-float  -mhard-float  -mmultiple  -mno-multiple
-mstring  -mno-string  -mupdate  -mno-update
-mfused-madd  -mno-fused-madd  -mbit-align  -mno-bit-align
-mstrict-align  -mno-strict-align  -mrelocatable
-mno-relocatable  -mrelocatable-lib  -mno-relocatable-lib
-mtoc  -mno-toc -mlittle  -mlittle-endian  -mbig  -mbig-endian
-mcall-aix  -mcall-sysv  -mprototype  -mno-prototype
-msim  -mmvme  -mads  -myellowknife  -memb -msdata
```
-msdata=*opt* -G *num*

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

MIPS Options
-mabicalls -mcpu=*cpu type* -membedded-data
```
-membedded-pic  -mfp32  -mfp64  -mgas  -mgp32  -mgp64
```

```
-mgpopt  -mhalf-pic  -mhard-float  -mint64  -mips1
-mips2  -mips3 -mips4 -mlong64  -mlong32 -mlong-calls  -mmemcpy
-mmips-as  -mmips-tfile  -mno-abicalls
-mno-embedded-data  -mno-embedded-pic
-mno-gpopt  -mno-long-calls
-mno-memcpy  -mno-mips-tfile  -mno-rnames  -mno-stats
-mrnames  -msoft-float
-m4650  -msingle-float  -mmad
-mstats  -EL  -EB  -G num  -nocpp
-mabi=32 -mabi=n32 -mabi=64 -mabi=eabi
```

i386 Options

```
-mcpu=cpu type
-march=cpu type
-mieee-fp  -mno-fancy-math-387
-mno-fp-ret-in-387  -msoft-float  -msvr3-shlib
-mno-wide-multiply  -mrtd  -malign-double
-mreg-alloc=list  -mregparm=num
-malign-jumps=num  -malign-loops=num
-malign-functions=num -mpreferred-stack-boundary=num
```

HPPA Options

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

Intel 960 Options

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

DEC Alpha Options

```
-mfp-regs  -mno-fp-regs -mno-soft-float  -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*
`-mbwx -mno-bwx -mcix -mno-cix -mmax -mno-max`
`-mmemory-latency=`*time*

Clipper Options
`-mc300  -mc400`

H8/300 Options
`-mrelax  -mh -ms -mint32  -malign-300`

SH Options
`-m1  -m2  -m3  -m3e  -mb  -ml  -mdalign -mrelax`

System V Options
`-Qy  -Qn  -YP,`*paths* `-Ym,`*dir*

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

TMS320C3x/C4x Options
`-mcpu=`*cpu* `-mbig -msmall -mregparm -mmemparm`
`-mfast-fix -mmpyi -mbk -mti -mdp-isr-reload`
`-mrpts=`*count* `-mrptb -mdb -mloop-unsigned`
`-mparallel-insns -mparallel-mpy -mpreserve-float`

V850 Options
`-mlong-calls -mno-long-calls -mep -mno-ep`
`-mprolog-function -mno-prolog-function -mspace`
`-mtda=`*n* `-msda=`*n* `-mzda=`*n*
`-mv850 -mbig-switch`

NS32K Options
`-m32032 -m32332 -m32532 -m32081 -m32381 -mmult-add -mnomult-add`
`-msoft-float -mrtd -mnortd -mregparam -mnoregparam -msb -mnosb`
`-mbitfield -mnobitfield -mhimem -mnohimem`

Code Generation Options
See Section 2.15 [Options for Code Generation Conventions], page 92.

`-fcall-saved-`*reg* `-fcall-used-`*reg*
`-fexceptions -ffixed-`*reg* `-finhibit-size-directive`
`-fcheck-memory-usage  -fprefix-function-name`
`-fno-common  -fno-ident  -fno-gnu-linker`
`-fpcc-struct-return  -fpic  -fPIC`
`-freg-struct-return  -fshared-data  -fshort-enums`
`-fshort-double  -fvolatile  -fvolatile-global -fvolatile-static`
`-fverbose-asm -fpack-struct  -fstack-check`
`-fargument-alias  -fargument-noalias`

```
-fargument-noalias-global
-fleading-underscore
```

2.2 Options Controlling the Kind of Output

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

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

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

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

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

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

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

file`.cc`
file`.cxx`
file`.cpp`
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`.s`
: Assembler code.

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

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

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

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

```
c  objective-c  c++
c-header  cpp-output  c++-cpp-output
assembler  assembler-with-cpp
```

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

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

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

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

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

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

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

Input files that don't require compilation are ignored.

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

Input files which don't require preprocessing are ignored.

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

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

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

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

`-pipe` Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

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

2.3 Compiling C++ Programs

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

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

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See Section 2.4 [Options Controlling C Dialect], page 12, for explanations of options for languages related to C. See Section 2.5 [Options Controlling C++ Dialect], page 16, for explanations of options that are meaningful only for C++ programs.

2.4 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 ANSI standard C programs. In C++ mode, remove GNU extensions that conflict with ANSI C++.

This turns off certain features of GCC that are incompatible with ANSI C (when compiling C code), or of ANSI 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 ANSI trigraph feature. For the C compiler, it disables recognition of C++ style '`//`' comments as well as the `inline` keyword. For the C++ compiler, '`-foperator-names`' is enabled as well.

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

The '`-ansi`' option does not cause non-ANSI programs to be rejected gratuitously. For that, '`-pedantic`' is required in addition to '`-ansi`'. See Section 2.6 [Warning Options], page 22.

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

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

`-fstd=` Determine the language standard. A value for this option must be provided; possible values are

- iso9899:1990 Same as -ansi

- iso9899:199409 ISO C as modified in amend. 1
- iso9899:199x ISO C 9x
- c89 same as -std=iso9899:1990
- c9x same as -std=iso9899:199x
- gnu89 default, iso9899:1990 + gnu extensions
- gnu9x iso9899:199x + gnu extensions

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 -fstd=c9x is not specified.

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

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

`-fno-builtin`

Don't recognize builtin functions that do not begin with '`__builtin_`' as prefix. Currently, the functions affected include `abort`, `abs`, `alloca`, `cos`, `exit`, `fabs`, `ffs`, `labs`, `memcmp`, `memcpy`, `sin`, `sqrt`, `strcmp`, `strcpy`, and `strlen`.

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

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

`-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`'.

`-trigraphs`

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

`-traditional`

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

- All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The newer keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)
- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
- Out-of-range floating point literals are not an error.
- Certain constructs which ANSI regards as a single invalid preprocessing number, such as '`0xe-0xd`', are treated as expressions instead.
- String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of '`-fwritable-strings`'.)
- All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.
- The character escape sequences '`\x`' and '`\a`' evaluate as the literal characters '`x`' and '`a`' respectively. Without '`-traditional`', '`\x`' is a prefix for the hexadecimal representation of a character, and '`\a`' produces a bell.

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

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

The '`-traditional`' option also enables '`-traditional-cpp`', which is described next.

`-traditional-cpp`

Attempt to support some aspects of traditional C preprocessors. Specifically:

- Comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In a preprocessing directive, the '`#`' symbol must appear as the first character of a line.
- Macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro `__STDC__` is not defined when you use '`-traditional`', but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by '`-traditional`'). If you need to write header files that

work differently depending on whether '`-traditional`' is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers. The predefined macro `__STDC_VERSION__` is also not defined when you use '`-traditional`'. See section "Standard Predefined Macros" in *The C Preprocessor*, for more discussion of these and other predefined macros.

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

`-fcond-mismatch`

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

`-funsigned-char`

Let the type `char` be unsigned, like `unsigned char`.

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

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

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

`-fsigned-char`

Let the type `char` be signed, like `signed char`.

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

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

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

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

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

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

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

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

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

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

2.5 Options Controlling C++ Dialect

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

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

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

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

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

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

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

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

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

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

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

`-fexternal-templates`
: Cause template instantiations to obey '`#pragma interface`' and '`implementation`'; template instances are emitted or not according to the location of the template definition. See Section 5.5 [Template Instantiation], page 180, for more information.

  This option is deprecated.

`-falt-external-templates`
: Similar to -fexternal-templates, but template instances are emitted or not according to the place where they are first instantiated. See Section 5.5 [Template Instantiation], page 180, for more information.

  This option is deprecated.

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

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

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

`-fguiding-decls`
: Treat a function declaration with the same type as a potential function template instantiation as though it declares that instantiation, not a normal function. If a definition is given for the function later in the translation unit (or another translation unit if the target supports weak symbols), that definition will be used; otherwise the template will be instantiated. This behavior reflects the C++ language prior to September 1996, when guiding declarations were removed.

This option implies '`-fname-mangling-version-0`', and will not work with other name mangling versions. Like all options that change the ABI, all C++ code, *including libgcc.a* must be built with the same setting of this option.

`-fhandle-signatures`
: Recognize the `signature` and `sigof` keywords for specifying abstract types. The default ('`-fno-handle-signatures`') is not to recognize them. See Section 5.7 [C++ Signatures], page 183.

`-fhonor-std`
: Treat the `namespace std` as a namespace, instead of ignoring it. For compatibility with earlier versions of g++, the compiler will, by default, ignore `namespace-declarations`, `using-declarations`, `using-directives`, and `namespace-names`, if they involve `std`.

`-fhuge-objects`
: Support virtual function calls for objects that exceed the size representable by a '`short int`'. Users should not use this flag by default; if you need to use it, the compiler will tell you so.

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

  Like all options that change the ABI, all C++ code, *including libgcc* must be built with the same setting of this option.

`-fno-implicit-templates`
: Never emit code for non-inline templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Section 5.5 [Template Instantiation], page 180, 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.

`-finit-priority`
: Support '`__attribute__ ((init_priority (n)))`' for controlling the order of initialization of file-scope objects. On ELF targets, this requires GNU ld 2.10 or later.

`-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.

`-fname-mangling-version-`*n*
: Control the way in which names are mangled. Version 0 is compatible with versions of g++ before 2.8. Version 1 is the default. Version 1 will allow correct mangling of function templates. For example, version 0 mangling does not mangle foo<int, double> and foo<int, char> given this declaration:

```
template <class T, class U> void foo(T t);
```

  Like all options that change the ABI, all C++ code, *including libgcc* must be built with the same setting of this option.

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

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

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

`-frepo`
: Enable automatic template instantiation. This option also implies '`-fno-implicit-templates`'. See Section 5.5 [Template Instantiation], page 180, for more information.

`-fno-rtti`
: Disable generation of the information used by C++ runtime type identification features ('`dynamic_cast`' and '`typeid`'). If you don't use those parts of the language (or exception handling, which uses '`dynamic_cast`' internally), you can save some space by using this flag.

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

  Specifying this option will also suppress implicit declarations of functions.

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

`-fsquangle`
`-fno-squangle`
: '`-fsquangle`' will enable a compressed form of name mangling for identifiers. In particular, it helps to shorten very long names by recognizing types and class names which occur more than once, replacing them with special short ID codes. This option also requires any C++ libraries being used to be compiled with this option as well. The compiler has this disabled (the equivalent of '`-fno-squangle`') by default.

  Like all options that change the ABI, all C++ code, *including libgcc.a* must be built with the same setting of this option.

`-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.

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

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

  The original implementation of thunks (version 1) had a bug regarding virtual base classes; this bug is fixed with version 2 of the thunks implementation. With setting the version to 2, compatibility to the version 1 thunks is provided, at the cost of extra machine code. Version 3 does not include this compatibility.

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

  Like all options that change the ABI, all C++ code, *including libgcc.a* must be built with the same setting of this option. Since version 1 and version 2 are also incompatible (for classes with virtual bases defining virtual functions), all code must also be compiled with the same version.

  In this version of gcc, there are no targets for which version 2 thunks are the default. On all targets, not giving the option will use the traditional implementation, and -fvtable-thunks will produce version 2 thunks.

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

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

`-fno-default-inline`
: Do not assume '`inline`' for functions defined inside a class scope. See Section 2.8 [Options That Control Optimization], page 35. Note that these functions will have linkage like inline functions; they just won't be inlined by default.

`-Wctor-dtor-privacy (C++ only)`
: Warn when a class seems unusable, because all the constructors or destructors in a class are private and the class has no friends or public static member functions.

`-Wnon-virtual-dtor (C++ only)`
: Warn when a class declares a non-virtual destructor that should probably be virtual, because it looks like the class will be used polymorphically.

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

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

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

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

`-Weffc++ (C++ only)`
: Warn about violations of various style guidelines from Scott Meyers' *Effective C++* books. If you use this option, you should be aware that the standard library headers do not obey all of these guidelines; you can use '`grep -v`' to filter out those warnings.

`-Wno-deprecated (C++ only)`
: Do not warn about usage of deprecated features. See Section 4.40 [Deprecated Features], page 176.

`-Wno-non-template-friend (C++ only)`
: Disable warnings when non-templatized friend functions are declared within a template. With the advent of explicit template specification support in g++, if the name of the friend is an unqualified-id (ie, '`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 also be turned off with the flag '`-fguiding-decls`', which activates the older, non-specification compiler code, or 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 is used within a C++ program. The new-style casts ('`static_cast`', '`reinterpret_cast`', and '`const_cast`') are less vulnerable to unintended effects.

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

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

`-Wsign-promo (C++ only)`
: Warn when overload resolution chooses a promotion from unsigned or enumeral type to a signed type over a conversion to an unsigned type of the same size. Previous versions of g++ would try to preserve unsignedness, but the standard mandates the current behavior.

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

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

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

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

2.6 Options to Request or Suppress Warnings

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

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

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

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

`-pedantic`
: Issue all the warnings demanded by strict ANSI C and ISO C++; reject all programs that use forbidden extensions.

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

'`-pedantic`' does not cause warning messages for use of the alternate keywords whose names begin and end with '`__`'. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them. See Section 4.35 [Alternate Keywords], page 173.

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

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

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

`-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.

`-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.

`-Wmultichar`
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.

`-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, GNU C 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 ();
    }
}
```

`-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`.

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

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

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

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

To suppress this warning for an expression, simply cast it to void. For unused variables, parameters and labels, use the '`unused`' attribute (see Section 4.29 [Variable Attributes], page 158).

`-Wuninitialized`

An automatic variable is used without first being initialized.

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

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

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

These warnings are made optional because 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.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 4.23 [Function Attributes], page 151.

`-Wunknown-pragmas`

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

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

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

-W Print extra warning messages for these events:

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

  The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.
- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

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

- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as '`x[i,j]`' will cause a warning, but '`x[(void)i,j]`' will not.
- An unsigned value is compared against zero with '`<`' or '`<=`'.
- A comparison like '`x<=y<=z`' appears; this is equivalent to '`(x<=y ? 1 : 0) <= z`', which is a different interpretation from that of ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- If '-Wall' or '-Wunused' is also specified, warn about unused arguments.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (But don't warn if '-Wno-sign-compare' is also specified.)
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

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

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

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

`-Wtraditional`
: Warn about certain constructs that behave differently in traditional and ANSI C.

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A `switch` statement has an operand of type `long`.
- A non-`static` function declaration follows a `static` one. This construct is not accepted by some traditional C compilers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`-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.

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

`-Wnested-externs`
: Warn if an `extern` declaration is encountered within an function.

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

`-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.

`-Werror`
: Make all warnings into errors.

2.7 Options for Debugging Your Program or GCC

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

`-g`
: Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

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

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

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

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

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

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

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

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

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

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

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

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

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

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gcoff`*level*
`-gxcoff`*level*
`-gdwarf`*level*
`-gdwarf-2`*level*
: Request debugging information and also use *level* to specify how much information. The default level is 2.

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

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

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

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

`-a`
: Generate extra code to write profile information for basic blocks, which will record the number of times each basic block is executed, the basic block start

address, and the function name containing the basic block. If '`-g`' is used, the line number and filename of the start of the basic block will also be recorded. If not overridden by the machine description, the default action is to append to the text file '`bb.out`'.

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

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

`-ax` Generate extra code to profile basic blocks. Your executable will produce output that is a superset of that produced when '`-a`' is used. Additional output is the source and target address of the basic blocks where a jump takes place, the number of times a jump is executed, and (optionally) the complete sequence of basic blocks being executed. The output is appended to file '`bb.out`'.

You can examine different profiling aspects without recompilation. Your executable will read a list of function names from file '`bb.in`'. Profiling starts when a function on the list is entered and stops when that invocation is exited. To exclude a function from profiling, prefix its name with '-'. If a function name is not unique, you can disambiguate it by writing it in the form '`/path/filename.d:functionname`'. Your executable will write the available paths and filenames in file '`bb.out`'.

Several function names have a special meaning:

`__bb_jumps__`
Write source, target and frequency of jumps to file '`bb.out`'.

`__bb_hidecall__`
Exclude function calls from frequency count.

`__bb_showret__`
Include function returns in frequency count.

`__bb_trace__`
Write the sequence of basic blocks executed to file '`bbtrace.gz`'. The file will be compressed using the program '`gzip`', which must exist in your `PATH`. On systems without the '`popen`' function, the file will be named '`bbtrace`' and will not be compressed. **Profiling for even a few seconds on these systems will produce a very large file.** Note: `__bb_hidecall__` and `__bb_showret__` will not affect the sequence written to '`bbtrace.gz`'.

Here's a short example using different profiling parameters in file '`bb.in`'. Assume function `foo` consists of basic blocks 1 and 2 and is called twice from block 3 of function `main`. After the calls, block 3 transfers control to block 4 of `main`.

With `__bb_trace__` and `main` contained in file '`bb.in`', the following sequence of blocks is written to file '`bbtrace.gz`': 0 3 1 2 1 2 4. The return from block 2 to block 3 is not shown, because the return is to a point inside the block and not to the top. The block address 0 always indicates, that control is transferred to the trace from somewhere outside the observed functions. With '`-foo`' added to '`bb.in`', the blocks of function `foo` are removed from the trace, so only 0 3 4 remains.

With `__bb_jumps__` and `main` contained in file '`bb.in`', jump frequencies will be written to file '`bb.out`'. The frequencies are obtained by constructing a trace of blocks and incrementing a counter for every neighbouring pair of blocks in the trace. The trace 0 3 1 2 1 2 4 displays the following frequencies:

```
Jump from block 0x0 to block 0x3 executed 1 time(s)
Jump from block 0x3 to block 0x1 executed 1 time(s)
Jump from block 0x1 to block 0x2 executed 2 time(s)
Jump from block 0x2 to block 0x1 executed 1 time(s)
Jump from block 0x2 to block 0x4 executed 1 time(s)
```

With `__bb_hidecall__`, control transfer due to call instructions is removed from the trace, that is the trace is cut into three parts: 0 3 4, 0 1 2 and 0 1 2. With `__bb_showret__`, control transfer due to return instructions is added to the trace. The trace becomes: 0 3 1 2 3 1 2 3 4. Note, that this trace is not the same, as the sequence written to '`bbtrace.gz`'. It is solely used for counting jump frequencies.

`-fprofile-arcs`

Instrument *arcs* during compilation. 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.

Since not every arc in the program must be instrumented, programs compiled with this option run faster than programs compiled with '`-a`', which adds instrumentation code to every basic block in the program. The tradeoff: since `gcov` does not have execution counts for all branches, it must start with the execution counts for the instrumented branches, and then iterate over the program flow graph until the entire graph has been solved. Hence, `gcov` runs a little more slowly than a program which uses information from '`-a`'.

'`-fprofile-arcs`' also makes it possible to estimate branch probabilities, and to calculate basic block execution counts. In general, basic block execution counts do not give enough information to estimate all branch probabilities. When the compiled program exits, it saves the arc execution counts to a file called '*sourcename*`.da`'. Use the compiler option '`-fbranch-probabilities`' (see Section 2.8 [Options that Control Optimization], page 35) when recompiling, to optimize using estimated branch probabilities.

`-ftest-coverage`
: Create data files for the `gcov` code-coverage utility (see Chapter 6 [`gcov`: a GCC Test Coverage Program], page 187). The data file names begin with the name of your source file:

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

  sourcename`.bbg`
  : A list of all arcs in the program flow graph. This allows `gcov` to reconstruct the program flow graph, so that it can compute all basic block and arc execution counts from the information in the *sourcename*`.da` file (this last file is the output from '`-fprofile-arcs`').

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

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

'`b`'	Dump after computing branch probabilities, to '*file*`.bp`'.
'`c`'	Dump after instruction combination, to the file '*file*`.combine`'.
'`d`'	Dump after delayed branch scheduling, to '*file*`.dbr`'.
'`D`'	Dump all macro definitions, at the end of preprocessing, in addition to normal output.
'`r`'	Dump after RTL generation, to '*file*`.rtl`'.
'`j`'	Dump after first jump optimization, to '*file*`.jump`'.
'`F`'	Dump after purging ADDRESSOF, to '*file*`.addressof`'.
'`f`'	Dump after flow analysis, to '*file*`.flow`'.
'`g`'	Dump after global register allocation, to '*file*`.greg`'.
'`G`'	Dump after GCSE, to '*file*`.gcse`'.
'`j`'	Dump after first jump optimization, to '*file*`.jump`'.
'`J`'	Dump after last jump optimization, to '*file*`.jump2`'.
'`k`'	Dump after conversion from registers to stack, to '*file*`.stack`'.
'`l`'	Dump after local register allocation, to '*file*`.lreg`'.
'`L`'	Dump after loop optimization, to '*file*`.loop`'.
'`M`'	Dump after performing the machine dependent reorganisation pass, to '*file*`.mach`'.

'`N`'	Dump after the register move pass, to '*file*`.regmove`'.
'`r`'	Dump after RTL generation, to '*file*`.rtl`'.
'`R`'	Dump after the second instruction scheduling pass, to '*file*`.sched2`'.
'`s`'	Dump after CSE (including the jump optimization that sometimes follows CSE), to '*file*`.cse`'.
'`S`'	Dump after the first instruction scheduling pass, to '*file*`.sched`'.
'`t`'	Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to '*file*`.cse2`'.
'`a`'	Produce all the dumps listed above.
'`m`'	Print statistics on memory usage, at the end of the run, to standard error.
'`p`'	Annotate the assembler output with a comment indicating which pattern and alternative was used. The length of each instruction is also printed.
'`x`'	Just generate RTL for a function instead of compiling it. Usually used with '`r`'.
'`y`'	Dump debugging information during parsing, to standard error.
'`A`'	Annotate the assembler output with miscellaneous debugging information.

`-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 invokations with different options, in particular with and without -g.

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

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

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

`-print-libgcc-file-name`

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

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

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

`-print-search-dirs`

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

This is useful when gcc prints the error message '`installation problem, cannot exec cpp: No such file or directory`'. To resolve this you either need to put '`cpp`' and the other compiler components where gcc expects to find them, or you can set the environment variable `GCC_EXEC_PREFIX` to the directory where you installed them. Don't forget the trailing '/'. See Section 2.16 [Environment Variables], page 97.

2.8 Options That Control Optimization

These options control various sorts of optimizations:

`-O`

`-O1`

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

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

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

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

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

`-O2`

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

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

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

`-O0` Do not optimize.

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

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

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

`-ffloat-store`

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

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have. Similarly for the x86 architecture. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '`-ffloat-store`' for such programs, after modifying them to store all pertinent intermediate computations into variables.

`-fno-default-inline`

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

`-fno-defer-pop`

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

`-fforce-mem`

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

`-fforce-addr`

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

`-fomit-frame-pointer`

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

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

`-fno-inline`

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

`-finline-functions`

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

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

`-finline-limit-`*n*

By default, gcc limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (ie marked with the inline keyword or defined within the class definition in c++). *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of n is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining heavily such as those based on recursive templates with c++.

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

`-fkeep-inline-functions`

Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function. This switch does not affect `extern inline` functions.

`-fkeep-static-consts`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`-fschedule-insns2`
: Similar to '`-fschedule-insns`', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`-fno-peephole`
: Disable any machine-specific peephole optimizations.

`-fbranch-probabilities`
: After running a program compiled with '`-fprofile-arcs`' (see Section 2.7 [Options for Debugging Your Program or `gcc`], page 29), you can compile it a second time using '`-fbranch-probabilities`', to improve optimizations based on guessing the path a branch might take.

`-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;
}
```

2.9 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.

`-include` *file*

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

`-imacros` *file*

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

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

'`-include`' and '`-imacros`' options are processed in the order in which they are written.

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

`-iprefix` *prefix*
: Specify *prefix* as the prefix for subsequent '`-iwithprefix`' options.

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

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

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

`-nostdinc`
: Do not search the standard system directories for header files. Only the directories you have specified with '`-I`' options (and the current directory, if appropriate) are searched. See Section 2.12 [Directory Options], page 46, for information on '`-I`'.

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

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

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

`-C`
: Tell the preprocessor not to discard comments. Used with the '`-E`' option.

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

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

  '`-M`' implies '`-E`'.

  Another way to specify output of a `make` rule is by setting the environment variable `DEPENDENCIES_OUTPUT` (see Section 2.16 [Environment Variables], page 97).

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

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

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

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

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

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

`-A`*question*`(`*answer*`)`
: Assert the answer *answer* for *question*, in case it is tested with a preprocessing conditional such as '`#if #`*question*`(`*answer*`)`'. '`-A-`' disables the standard assertions that normally describe the target machine.

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

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

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

`-dM`
: Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the '`-E`' option.

`-dD`
: Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output.

`-dN`
: Like '`-dD`' except that the macro arguments and contents are omitted. Only '`#define` *name*' is included in the output.

`-trigraphs`
: Support ANSI C trigraphs. The '`-ansi`' option also has this effect.

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

2.10 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.

2.11 Options for Linking

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

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

`-c`
`-S`
`-E`
: If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See Section 2.2 [Overall Options], page 10.

`-l`*library*
: Search the library named *library* when linking.

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

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

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

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

`-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 ANSI 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 ANSI C) environments or to bcopy and bzero for BSD environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

  One of the standard libraries bypassed by '`-nostdlib`' and '`-nodefaultlibs`' is '`libgcc.a`', a library of internal subroutines that GCC uses to overcome shortcomings of particular machines, or special needs for some languages. (See section "Interfacing to GCC Output" in *Porting GCC*, for more discussion of '`libgcc.a`'.) In most cases, you need '`libgcc.a`' even when you want to avoid other standard libraries. In other words, when you specify '`-nostdlib`' or '`-nodefaultlibs`' you should usually specify '`-lgcc`' as well. This ensures that you have no unresolved references to internal GCC library subroutines. (For example, '`__main`', used to ensure C++ constructors will be called; see Section 3.7 [`collect2`], page 134.)

`-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. You must also specify '`-fpic`' or '`-fPIC`' on some systems when you specify this option.

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

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

2.12 Options for Directory Search

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

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

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

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

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

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

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

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

  The compiler driver program runs one or more of the subprograms '`cpp`', '`cc1`', '`as`' and '`ld`'. It tries *prefix* as a prefix for each program it tries to run, both with and without '*machine/version/*' (see Section 2.13 [Target Options], page 47).

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

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

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

Another way to specify a prefix much like the '`-B`' prefix is to use the environment variable `GCC_EXEC_PREFIX`. See Section 2.16 [Environment Variables], page 97.

`-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.

2.13 Specifying Target Machine and Compiler Version

By default, GCC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GCC, for different target machines, can be installed side by side. Then you specify which one to use with the '`-b`' option.

In addition, older and newer versions of GCC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

`-b` *machine*
: The argument *machine* specifies the target machine for compilation. This is useful when you have installed GCC as a cross-compiler.

  The value to use for *machine* is the same as was specified as the machine type when configuring GCC as a cross-compiler. For example, if a cross-compiler was configured with '`configure i386v`', meaning to compile for an 80386 running System V, then you would specify '`-b i386v`' to run that cross compiler.

  When you do not specify '`-b`', it normally means to compile for the same type of machine that you are using.

`-V` *version*
: The argument *version* specifies which version of GCC to run. This is useful when multiple versions are installed. For example, *version* might be '`2.0`', meaning to run GCC version 2.0.

  The default version, when you do not specify '`-V`', is the last version of GCC that you installed.

The '`-b`' and '`-V`' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GCC, for a given target machine, is normally kept in the directory '`/usr/local/lib/gcc-lib/`*machine*`/`*version*'.

Thus, sites can customize the effect of '`-b`' or '`-V`' either by changing the names of these directories or adding alternate names (or symbolic links). If in directory '`/usr/local/lib/gcc-lib/`' the file '`80386`' is a link to the file '`i386v`', then '`-b 80386`' becomes an alias for '`-b i386v`'.

In one respect, the '`-b`' or '`-V`' do not completely change to a different compiler: the top-level driver program `gcc` that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target and version.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine and compiler version.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as `ogcc` and that for version 2.1 is installed as `gcc`, then the command `gcc` will use version 2.1 by default, while `ogcc` will use 2.0 by default. However, you can choose either version with either command with the '`-V`' option.

2.14 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.

2.14.1 M680x0 Options

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

`-m68000`
`-mc68000`
: Generate output for a 68000. This is the default when the compiler is configured for 68000-based systems.

  Use this option for microcontrollers with a 68000 or EC000 core, including the 68008, 68302, 68306, 68307, 68322, 68328 and 68356.

`-m68020`
`-mc68020`
: Generate output for a 68020. This is the default when the compiler is configured for 68020-based systems.

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

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

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

This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. Use this option if your 68040 does not have code to emulate those instructions.

`-m68060`
: Generate output for a 68060. This is the default when the compiler is configured for 68060-based systems.

  This option inhibits the use of 68020 and 68881/68882 instructions that have to be emulated by software on the 68060. Use this option if your 68060 does not have code to emulate those instructions.

`-mcpu32`
: Generate output for a CPU32. This is the default when the compiler is configured for CPU32-based systems.

  Use this option for microcontrollers with a CPU32 or CPU32+ core, including the 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349 and 68360.

`-m5200`
: Generate output for a 520X "coldfire" family cpu. This is the default when the compiler is configured for 520X-based systems.

  Use this option for microcontroller with a 5200 core, including the MCF5202, MCF5203, MCF5204 and MCF5202.

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

`-m68020-60`
: Generate output for a 68060, without using any of the new instructions. This results in code which can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68060.

`-mfpa`
: Generate output containing Sun FPA instructions for floating point.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all m68k targets. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets '`m68k-*-aout`' and '`m68k-*-coff`' do provide software floating point support.

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

`-mnobitfield`
: Do not use the bit-field instructions. The '`-m68000`', '`-mcpu32`' and '`-m5200`' options imply '`-mnobitfield`'.

`-mbitfield`
: Do use the bit-field instructions. The '`-m68020`' option implies '`-mbitfield`'. This is the default if you use a configuration designed for a 68020.

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

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

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

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

  The `rtd` instruction is supported by the 68010, 68020, 68030, 68040, 68060 and CPU32 processors, but not by the 68000 or 5200.

`-malign-int`
`-mno-align-int`
: Control whether GCC aligns `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 32-bit boundary (‘`-malign-int`’) or a 16-bit boundary (‘`-mno-align-int`’). Aligning variables on 32-bit boundaries produces code that runs somewhat faster on processors with 32-bit busses at the expense of more memory.

  Warning: if you use the ‘`-malign-int`’ switch, GCC will align structures containing the above types differently than most published application binary interface specifications for the m68k.

2.14.2 VAX Options

These ‘`-m`’ options are defined for the Vax:

`-munix`
: Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the Vax cannot handle across long ranges.

`-mgnu`
: Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

`-mg`
: Output code for g-format floating point numbers instead of d-format.

2.14.3 SPARC Options

These ‘`-m`’ switches are supported on the SPARC:

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

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

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

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

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

`-mhard-quad-float`
: Generate output containing quad-word (long double) floating point instructions.

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

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

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

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

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

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

`-mno-unaligned-doubles`
`-munaligned-doubles`

Assume that doubles have 8 byte alignment. This is the default.

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

`-mv8`
`-msparclite`

These two options select variations on the SPARC architecture.

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

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

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

These options are deprecated and will be deleted in a future GCC release. They have been replaced with '`-mcpu=xxx`'.

`-mcypress`
`-msupersparc`

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

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

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

These options are deprecated and will be deleted in a future GCC release. They have been replaced with '`-mcpu=xxx`'.

`-mcpu=`*cpu_type*

Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`v7`', '`cypress`', '`v8`', '`supersparc`', '`sparclite`', '`hypersparc`', '`sparclite86x`', '`f930`', '`f934`', '`sparclet`', '`tsc701`', '`v9`', and '`ultrasparc`'.

Default instruction scheduling parameters are used for values that select an architecture and not an implementation. These are '`v7`', '`v8`', '`sparclite`', '`sparclet`', '`v9`'.

Here is a list of each supported architecture and their supported implementations.

`v7:`	`cypress`
`v8:`	`supersparc, hypersparc`
`sparclite:`	`f930, f934, sparclite86x`
`sparclet:`	`tsc701`
`v9:`	`ultrasparc`

`-mtune=`*cpu_type*
: Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the instruction set or register set that the option '`-mcpu=`'*cpu_type* would.

  The same values for '`-mcpu=`'*cpu_type* are used for '`-mtune=`' *cpu_type*, though the only useful values are those that select a particular cpu implementation: '`cypress`', '`supersparc`', '`hypersparc`', '`f930`', '`f934`', '`sparclite86x`', '`tsc701`', '`ultrasparc`'.

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

`-malign-jumps=`*num*
: Align instructions that are only jumped to to a 2 raised to a *num* byte boundary. If '`-malign-jumps`' is not specified, the default is 2.

`-malign-functions=`*num*
: Align the start of functions to a 2 raised to *num* byte boundary. If '`-malign-functions`' is not specified, the default is 2 if compiling for 32 bit sparc, and 5 if compiling for 64 bit sparc.

These '`-m`' switches are supported in addition to the above on the SPARCLET processor.

`-mlittle-endian`
: Generate code for a processor running in little-endian mode.

`-mlive-g0`
: Treat register `%g0` as a normal register. GCC will continue to clobber it as necessary but will not assume it always reads as 0.

`-mbroken-saverestore`
: Generate code that does not use non-trivial forms of the `save` and `restore` instructions. Early versions of the SPARCLET processor do not correctly handle `save` and `restore` instructions used with arguments. They correctly handle them used without arguments. A `save` instruction used without arguments increments the current window pointer but does not allocate a new stack frame. It is assumed that the window overflow trap handler will properly handle this case as will interrupt handlers.

These '`-m`' switches are supported in addition to the above on SPARC V9 processors in 64 bit environments.

`-mlittle-endian`
: Generate code for a processor running in little-endian mode.

`-m32`
`-m64`
: Generate code for a 32 bit or 64 bit environment. The 32 bit environment sets int, long and pointer to 32 bits. The 64 bit environment sets int to 32 bits and long and pointer to 64 bits.

`-mcmodel=medlow`
: Generate code for the Medium/Low code model: the program must be linked in the low 32 bits of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked.

`-mcmodel=medmid`
: Generate code for the Medium/Middle code model: the program must be linked in the low 44 bits of the address space, the text segment must be less than 2G bytes, and data segment must be within 2G of the text segment. Pointers are 64 bits.

`-mcmodel=medany`
: Generate code for the Medium/Anywhere code model: the program may be linked anywhere in the address space, the text segment must be less than 2G bytes, and data segment must be within 2G of the text segment. Pointers are 64 bits.

`-mcmodel=embmedany`
: Generate code for the Medium/Anywhere code model for embedded systems: assume a 32 bit text and a 32 bit data segment, both starting anywhere (determined at link time). Register %g4 points to the base of the data segment. Pointers still 64 bits. Programs are statically linked, PIC is not supported.

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

2.14.4 Convex Options

These '`-m`' options are defined for Convex:

`-mc1`
: Generate output for C1. The code will run on any Convex machine. The preprocessor symbol `__convex__c1__` is defined.

`-mc2`
: Generate output for C2. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C2. The preprocessor symbol `__convex_c2__` is defined.

`-mc32`
: Generate output for C32xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C32. The preprocessor symbol `__convex_c32__` is defined.

`-mc34`
: Generate output for C34xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C34. The preprocessor symbol `__convex_c34__` is defined.

`-mc38` Generate output for C38xx. Uses instructions not available on C1. Scheduling and other optimizations are chosen for max performance on C38. The preprocessor symbol `__convex_c38__` is defined.

`-margcount`
Generate code which puts an argument count in the word preceding each argument list. This is compatible with regular CC, and a few programs may need the argument count word. GDB and other source-level debuggers do not need it; this info is in the symbol table.

`-mnoargcount`
Omit the argument count word. This is the default.

`-mvolatile-cache`
Allow volatile references to be cached. This is the default.

`-mvolatile-nocache`
Volatile references bypass the data cache, going all the way to memory. This is only needed for multi-processor code that does not use standard synchronization instructions. Making non-volatile references to volatile locations will not necessarily work.

`-mlong32` Type long is 32 bits, the same as type int. This is the default.

`-mlong64` Type long is 64 bits, the same as type long long. This option is useless, because no library support exists for it.

2.14.5 AMD29K Options

These '`-m`' options are defined for the AMD Am29000:

`-mdw` Generate code that assumes the `DW` bit is set, i.e., that byte and halfword operations are directly supported by the hardware. This is the default.

`-mndw` Generate code that assumes the `DW` bit is not set.

`-mbw` Generate code that assumes the system supports byte and halfword write operations. This is the default.

`-mnbw` Generate code that assumes the systems does not support byte and halfword write operations. '`-mnbw`' implies '`-mndw`'.

`-msmall` Use a small memory model that assumes that all function addresses are either within a single 256 KB segment or at an absolute address of less than 256k. This allows the `call` instruction to be used instead of a `const`, `consth`, `calli` sequence.

`-mnormal` Use the normal memory model: Generate `call` instructions only when calling functions in the same file and `calli` instructions otherwise. This works if each file occupies less than 256 KB but allows the entire executable to be larger than 256 KB. This is the default.

`-mlarge` Always use `calli` instructions. Specify this option if you expect a single file to compile into more than 256 KB of code.

`-m29050` Generate code for the Am29050.

`-m29000` Generate code for the Am29000. This is the default.

`-mkernel-registers`
: Generate references to registers `gr64-gr95` instead of to registers `gr96-gr127`. This option can be used when compiling kernel code that wants a set of global registers disjoint from that used by user-mode code.

  Note that when this option is used, register names in '`-f`' flags must use the normal, user-mode, names.

`-muser-registers`
: Use the normal set of global registers, `gr96-gr127`. This is the default.

`-mstack-check`
`-mno-stack-check`
: Insert (or do not insert) a call to `__msp_check` after each stack adjustment. This is often used for kernel code.

`-mstorem-bug`
`-mno-storem-bug`
: '`-mstorem-bug`' handles 29k processors which cannot handle the separation of a mtsrim insn and a storem instruction (most 29000 chips to date, but not the 29050).

`-mno-reuse-arg-regs`
`-mreuse-arg-regs`
: '`-mno-reuse-arg-regs`' tells the compiler to only use incoming argument registers for copying out arguments. This helps detect calling a function with fewer arguments than it was declared with.

`-mno-impure-text`
`-mimpure-text`
: '`-mimpure-text`', used in addition to '`-shared`', tells the compiler to not pass '`-assert pure-text`' to the linker when linking a shared object.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mno-multm`
: Do not generate multm or multmu instructions. This is useful for some embedded systems which do not have trap handlers for these instructions.

2.14.6 ARM Options

These '`-m`' options are defined for Advanced RISC Machines (ARM) architectures:

`-mapcs-frame`
: Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying '`-fomit-frame-pointer`' with this option will cause the stack frames not to be generated for leaf functions. The default is '`-mno-apcs-frame`'.

`-mapcs`
: This is a synonym for '`-mapcs-frame`'.

`-mapcs-26`
: Generate code for a processor running with a 26-bit program counter, and conforming to the function calling standards for the APCS 26-bit option. This option replaces the '`-m2`' and '`-m3`' options of previous releases of the compiler.

`-mapcs-32`
: Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option. This option replaces the '`-m6`' option of previous releases of the compiler.

`-mapcs-stack-check`
: Generate code to check the amount of stack space available upon entry to every function (that actually uses some stack space). If there is insufficient space available then either the function '`__rt_stkovf_split_small`' or '`__rt_stkovf_split_big`' will be called, depending upon the amount of stack space required. The run time system is required to provide these functions. The default is '`-mno-apcs-stack-check`', since this produces smaller code.

`-mapcs-float`
: Pass floating point arguments using the float point registers. This is one of the variants of the APCS. This option is reccommended if the target hardware has a floating point unit or if a lot of floating point arithmetic is going to be performed by the code. The default is '`-mno-apcs-float`', since integer only code is slightly increased in size if '`-mapcs-float`' is used.

`-mapcs-reentrant`
: Generate reentrant, position independent code. This is the equivalent to specifying the '`-fpic`' option. The default is '`-mno-apcs-reentrant`'.

`-mthumb-interwork`
: Generate code which supports calling between the ARM and THUMB instruction sets. Without this option the two instruction sets cannot be reliably used inside one program. The default is '`-mno-thumb-interwork`', since slightly larger code is generated when '`-mthumb-interwork`' is specified.

`-mno-sched-prolog`
: Prevent the reordering of instructions in the function prolog, or the merging of those instruction with the instructions in the function's body. This means that all functions will start with a recognisable set of instructions (or in fact one of a chioce from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is '`-msched-prolog`'.

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

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all ARM targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

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

`-mlittle-endian`
: Generate code for a processor running in little-endian mode. This is the default for all standard configurations.

`-mbig-endian`
: Generate code for a processor running in big-endian mode; the default is to compile code for a little-endian processor.

`-mwords-little-endian`
: This option only applies when generating code for big-endian processors. Generate code for a little-endian word order but a big-endian byte order. That is, a byte order of the form '`32107654`'. Note: this option should only be used if you require compatibility with code for big-endian ARM processors generated by versions of the compiler prior to 2.8.

`-mshort-load-bytes`
: Do not try to load half-words (eg '`short`'s) by loading a word from an unaligned address. For some targets the MMU is configured to trap unaligned loads; use this option to generate code that is safe in these environments.

`-mno-short-load-bytes`
: Use unaligned word loads to load half-words (eg '`short`'s). This option produces more efficient code, but the MMU is sometimes configured to trap these instructions.

`-mshort-load-words`
: This is a synonym for the '`-mno-short-load-bytes`'.

`-mno-short-load-words`
: This is a synonym for the '`-mshort-load-bytes`'.

`-mbsd`
: This option only applies to RISC iX. Emulate the native BSD-mode compiler. This is the default if '`-ansi`' is not specified.

`-mxopen`
: This option only applies to RISC iX. Emulate the native X/Open-mode compiler.

`-mno-symrename`
: This option only applies to RISC iX. Do not run the assembler post-processor, '`symrename`', after code has been assembled. Normally it is necessary to modify

some of the standard symbols in preparation for linking with the RISC iX C library; this option suppresses this pass. The post-processor is never run when the compiler is built for cross-compilation.

`-mcpu=<name>`
`-mtune=<name>`
: This specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can use when generating assembly code. Permissable names are: arm2, arm250, arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m, arm7d, arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710c, arm7100, arm7500, arm7500fe, arm7tdmi, arm8, strongarm, strongarm110, strongarm1100, arm8, arm810, arm9, arm9tdmi. '`-mtune=`' is a synonym for '`-mcpue=`' to support older versions of GCC.

`-march=<name>`
: This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can use when generating assembly code. This option can be used in conjunction with or instead of the '`-mcpu=`' option. Permissable names are: armv2, armv2a, armv3, armv3m, armv4, armv4t

`-mfpe=<number>`
`-mfp=<number>`
: This specifes the version of the floating point emulation available on the target. Permissable values are 2 and 3. '`-mfp=`' is a synonym for '`-mfpe=`' to support older versions of GCC.

`-mstructure-size-boundary=<n>`
: The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissable values are 8 and 32. The default value varies for different toolchains. For the COFF targeted toolchain the default value is 8. Specifying the larger number can produced faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with the other value, if they exchange information using structures or unions. Programmers are encouraged to use the 32 value as future versions of the toolchain may default to this value.

`-mabort-on-noreturn`
: Generate a call to the function abort at the end of a noreturn function. It will be executed if the function tries to return.

2.14.7 Thumb Options

`-mthumb-interwork`
: Generate code which supports calling between the THUMB and ARM instruction sets. Without this option the two instruction sets cannot be reliably used inside one program. The default is '`-mno-thumb-interwork`', since slightly smaller code is generated with this option.

`-mtpcs-frame`
: Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all non-leaf functions. (A leaf function is one that does not call any other functions). The default is '`-mno-apcs-frame`'.

`-mtpcs-leaf-frame`
: Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all leaf functions. (A leaf function is one that does not call any other functions). The default is '`-mno-apcs-leaf-frame`'.

`-mlittle-endian`
: Generate code for a processor running in little-endian mode. This is the default for all standard configurations.

`-mbig-endian`
: Generate code for a processor running in big-endian mode.

`-mstructure-size-boundary=<n>`
: The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissable values are 8 and 32. The default value varies for different toolchains. For the COFF targeted toolchain the default value is 8. Specifying the larger number can produced faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with the other value, if they exchange information using structures or unions. Programmers are encouraged to use the 32 value as future versions of the toolchain may default to this value.

2.14.8 MN10200 Options

These '`-m`' options are defined for Matsushita MN10200 architectures:

`-mrelax`
: Indicate to the linker that it should perform a relaxation optimization pass to shorten branches, calls and absolute memory addresses. This option only has an effect when used on the command line for the final link step.

  This option makes symbolic debugging impossible.

2.14.9 MN10300 Options

These '`-m`' options are defined for Matsushita MN10300 architectures:

`-mmult-bug`
: Generate code to avoid bugs in the multiply instructions for the MN10300 processors. This is the default.

`-mno-mult-bug`
: Do not generate code to avoid bugs in the multiply instructions for the MN10300 processors.

`-mrelax`
: Indicate to the linker that it should perform a relaxation optimization pass to shorten branches, calls and absolute memory addresses. This option only has an effect when used on the command line for the final link step.

  This option makes symbolic debugging impossible.

2.14.10 M32R/D Options

These '`-m`' options are defined for Mitsubishi M32R/D architectures:

`-mcode-model=small`
: Assume all objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and assume all subroutines are reachable with the `bl` instruction. This is the default.

  The addressability of a particular object can be set with the `model` attribute.

`-mcode-model=medium`
: Assume objects may be anywhere in the 32 bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and assume all subroutines are reachable with the `bl` instruction.

`-mcode-model=large`
: Assume objects may be anywhere in the 32 bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and assume subroutines may not be reachable with the `bl` instruction (the compiler will generate the much slower `seth/add3/jl` instruction sequence).

`-msdata=none`
: Disable use of the small data area. Variables will be put into one of '`.data`', '`bss`', or '`.rodata`' (unless the `section` attribute has been specified). This is the default.

  The small data area consists of sections '`.sdata`' and '`.sbss`'. Objects may be explicitly put in the small data area with the `section` attribute using one of these sections.

`-msdata=sdata`
: Put small global and static data in the small data area, but do not generate special code to reference them.

`-msdata=use`
: Put small global and static data in the small data area, and generate special instructions to reference them.

`-G` *num*
: Put global and static objects less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss sections. The default value of *num* is 8. The '`-msdata`' option must be set to one of '`sdata`' or '`use`' for this option to have any effect.

  All modules should be compiled with the same '`-G` *num*' value. Compiling with different values of *num* may or may not work; if it doesn't the linker will give an error message - incorrect code will not be generated.

2.14.11 M88K Options

These '`-m`' options are defined for Motorola 88k architectures:

`-m88000`
: Generate code that works well on both the m88100 and the m88110.

`-m88100` Generate code that works best for the m88100, but that also runs on the m88110.

`-m88110` Generate code that works best for the m88110, and may not run on the m88100.

`-mbig-pic`
: Obsolete option to be removed from the next revision. Use '`-fPIC`'.

`-midentify-revision`
: Include an `ident` directive in the assembler output recording the source file name, compiler name and version, timestamp, and compilation flags used.

`-mno-underscores`
: In assembler output, emit symbol names without adding an underscore character at the beginning of each name. The default is to use an underscore as prefix on each name.

`-mocs-debug-info`
`-mno-ocs-debug-info`
: Include (or omit) additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard, "OCS". This extra information allows debugging of code that has had the frame pointer eliminated. The default for DG/UX, SVr4, and Delta 88 SVr3.2 is to include this information; other 88k configurations omit this information by default.

`-mocs-frame-position`
: When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address, which is the stack pointer (register 31) on entry to the function. The DG/UX, SVr4, Delta88 SVr3.2, and BCS configurations use '`-mocs-frame-position`'; other 88k configurations have the default '`-mno-ocs-frame-position`'.

`-mno-ocs-frame-position`
: When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the -g switch.

`-moptimize-arg-area`
`-mno-optimize-arg-area`
: Control how function arguments are stored in stack frames. '`-moptimize-arg-area`' saves space by optimizing them, but this conflicts with the 88open specifications. The opposite alternative, '`-mno-optimize-arg-area`', agrees with 88open standards. By default GCC does not optimize the argument area.

`-mshort-data-`*num*
: Generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). You control which data references are affected by specifying *num* with this option. For example, if you specify '`-mshort-data-512`', then the data references affected are those involving displacements of less than 512 bytes. '`-mshort-data-`*num*' is not effective for *num* greater than 64k.

`-mserialize-volatile`
`-mno-serialize-volatile`
: Do, or don't, generate code to guarantee sequential consistency of volatile memory references. By default, consistency is guaranteed.

  The order of memory references made by the MC88110 processor does not always match the order of the instructions requesting those references. In particular, a load instruction may execute before a preceding store instruction. Such reordering violates sequential consistency of volatile memory references, when there are multiple processors. When consistency must be guaranteed, GNU C generates special instructions, as needed, to force execution in the proper order.

  The MC88100 processor does not reorder memory references and so always provides sequential consistency. However, by default, GNU C generates the special instructions to guarantee consistency even when you use '`-m88100`', so that the code may be run on an MC88110 processor. If you intend to run your code only on the MC88100 processor, you may use '`-mno-serialize-volatile`'.

  The extra code generated to guarantee consistency may affect the performance of your application. If you know that you can safely forgo this guarantee, you may use '`-mno-serialize-volatile`'.

`-msvr4`
`-msvr3`
: Turn on ('`-msvr4`') or off ('`-msvr3`') compiler extensions related to System V release 4 (SVr4). This controls the following:

  1. Which variant of the assembler syntax to emit.
  2. '`-msvr4`' makes the C preprocessor recognize '`#pragma weak`' that is used on System V release 4.
  3. '`-msvr4`' makes GCC issue additional declaration directives used in SVr4.

  '`-msvr4`' is the default for the m88k-motorola-sysv4 and m88k-dg-dgux m88k configurations. '`-msvr3`' is the default for all other m88k configurations.

`-mversion-03.00`
: This option is obsolete, and is ignored.

`-mno-check-zero-division`
`-mcheck-zero-division`
: Do, or don't, generate code to guarantee that integer division by zero will be detected. By default, detection is guaranteed.

  Some models of the MC88100 processor fail to trap upon integer division by zero under certain conditions. By default, when compiling code that might be run on such a processor, GNU C generates code that explicitly checks for zero-valued divisors and traps with exception number 503 when one is detected. Use of mno-check-zero-division suppresses such checking for code generated to run on an MC88100 processor.

  GNU C assumes that the MC88110 processor correctly detects all instances of integer division by zero. When '`-m88110`' is specified, both '`-mcheck-zero-division`' and '`-mno-check-zero-division`' are ignored, and no explicit checks for zero-valued divisors are generated.

`-muse-div-instruction`

Use the div instruction for signed integer division on the MC88100 processor. By default, the div instruction is not used.

On the MC88100 processor the signed integer division instruction div) traps to the operating system on a negative operand. The operating system transparently completes the operation, but at a large cost in execution time. By default, when compiling code that might be run on an MC88100 processor, GNU C emulates signed integer division using the unsigned integer division instruction divu), thereby avoiding the large penalty of a trap to the operating system. Such emulation has its own, smaller, execution cost in both time and space. To the extent that your code's important signed integer division operations are performed on two nonnegative operands, it may be desirable to use the div instruction directly.

On the MC88110 processor the div instruction (also known as the divs instruction) processes negative operands without trapping to the operating system. When '`-m88110`' is specified, '`-muse-div-instruction`' is ignored, and the div instruction is used for signed integer division.

Note that the result of dividing INT_MIN by -1 is undefined. In particular, the behavior of such a division with and without '`-muse-div-instruction`' may differ.

`-mtrap-large-shift`
`-mhandle-large-shift`

Include code to detect bit-shifts of more than 31 bits; respectively, trap such shifts or emit code to handle them properly. By default GCC makes no special provision for large bit shifts.

`-mwarn-passed-structs`

Warn when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of portability problems. By default, GCC issues no such warning.

2.14.12 IBM RS/6000 and PowerPC Options

These '`-m`' options are defined for the IBM RS/6000 and PowerPC:

`-mpower`
`-mno-power`
`-mpower2`
`-mno-power2`
`-mpowerpc`
`-mno-powerpc`
`-mpowerpc-gpopt`
`-mno-powerpc-gpopt`
`-mpowerpc-gfxopt`
`-mno-powerpc-gfxopt`
`-mpowerpc64`
`-mno-powerpc64`

GCC supports two related instruction set architectures for the RS/6000 and PowerPC. The *POWER* instruction set are those instructions supported by the '`rios`' chip set used in the original RS/6000 systems and the *PowerPC* instruction set is the architecture of the Motorola MPC5xx, MPC6xx, MPC8xx microprocessors, and the IBM 4xx microprocessors.

Neither architecture is a subset of the other. However there is a large common subset of instructions supported by both. An MQ register is included in processors supporting the POWER architecture.

You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GCC. Specifying the '`-mcpu=`*cpu_type*' overrides the specification of these options. We recommend you use the '`-mcpu=`*cpu_type*' option rather than the options listed above.

The '`-mpower`' option allows GCC to generate instructions that are found only in the POWER architecture and to use the MQ register. Specifying '`-mpower2`' implies '`-power`' and also allows GCC to generate instructions that are present in the POWER2 architecture but not the original POWER architecture.

The '`-mpowerpc`' option allows GCC to generate instructions that are found only in the 32-bit subset of the PowerPC architecture. Specifying '`-mpowerpc-gpopt`' implies '`-mpowerpc`' and also allows GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying '`-mpowerpc-gfxopt`' implies '`-mpowerpc`' and also allows GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

The '`-mpowerpc64`' option allows GCC to generate the additional 64-bit instructions that are found in the full PowerPC64 architecture and to treat GPRs as 64-bit, doubleword quantities. GCC defaults to '`-mno-powerpc64`'.

If you specify both '`-mno-power`' and '`-mno-powerpc`', GCC will use only the instructions in the common subset of both architectures plus some special AIX common-mode calls, and will not use the MQ register. Specifying both '`-mpower`' and '`-mpowerpc`' permits GCC to use any instruction from either architecture and to allow use of the MQ register; specify this for the Motorola MPC601.

`-mnew-mnemonics`
`-mold-mnemonics`

Select which mnemonics to use in the generated assembler code. '`-mnew-mnemonics`' requests output that uses the assembler mnemonics defined for the PowerPC architecture, while '`-mold-mnemonics`' requests 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 '`rs6000`', '`rios1`', '`rios2`', '`rsc`', '`601`', '`602`', '`603`', '`603e`', '`604`', '`604e`', '`620`', '`740`', '`750`', '`power`', '`power2`', '`powerpc`', '`403`', '`505`', '`801`', '`821`', '`823`', and '`860`' and '`common`'. '`-mcpu=power`', '`-mcpu=power2`', and '`-mcpu=powerpc`' specify generic POWER, POWER2 and pure PowerPC (i.e., not MPC601) architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Specifying any of the following options: '`-mcpu=rios1`', '`-mcpu=rios2`', '`-mcpu=rsc`', '`-mcpu=power`', or '`-mcpu=power2`' enables the '`-mpower`' option and disables the '`-mpowerpc`' option; '`-mcpu=601`' enables both the '`-mpower`' and '`-mpowerpc`' options. All of '`-mcpu=602`', '`-mcpu=603`', '`-mcpu=603e`', '`-mcpu=604`', '`-mcpu=620`', enable the '`-mpowerpc`' option and disable the '`-mpower`' option. Exactly similarly, all of '`-mcpu=403`', '`-mcpu=505`', '`-mcpu=821`', '`-mcpu=860`' and '`-mcpu=powerpc`' enable the '`-mpowerpc`' option and disable the '`-mpower`' option. '`-mcpu=common`' disables both the '`-mpower`' and '`-mpowerpc`' options.

AIX versions 4 or greater selects '`-mcpu=common`' by default, so that code will operate on all members of the RS/6000 and PowerPC families. In that case, 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. GCC assumes a generic processor model for scheduling purposes.

Specifying any of the options '`-mcpu=rios1`', '`-mcpu=rios2`', '`-mcpu=rsc`', '`-mcpu=power`', or '`-mcpu=power2`' also disables the '`new-mnemonics`' option. Specifying '`-mcpu=601`', '`-mcpu=602`', '`-mcpu=603`', '`-mcpu=603e`', '`-mcpu=604`', '`620`', '`403`', or '`-mcpu=powerpc`' also enables the '`new-mnemonics`' option.

Specifying '`-mcpu=403`', '`-mcpu=821`', or '`-mcpu=860`' also enables the '`-msoft-float`' option.

`-mtune=`*cpu_type*

Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the architecture type, register usage, choice of mnemonics like '`-mcpu=`'*cpu_type* would. The same values for *cpu_type* are used for '`-mtune=`'*cpu_type* as for '`-mcpu=`'*cpu_type*. The '`-mtune=`'*cpu_type* option overrides the '`-mcpu=`'*cpu_type* option in terms of instruction scheduling parameters.

`-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 AIX 64-bit 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.

`-mthreads`

Support *AIX Threads*. Link an application written to use *pthreads* with special libraries and startup code to enable the application to run.

`-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 '`-mthreads`' option are incompatible.

`-msoft-float`
`-mhard-float`
Generate code that does not use (uses) the floating-point register set. Software floating point emulation is provided if you use the '`-msoft-float`' option, and pass the option to GCC when linking.

`-mmultiple`
`-mno-multiple`
Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '`-mmultiple`' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750 which permit the instructions usage in little endian mode.

`-mstring`
`-mno-string`
Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '`-mstring`' on little endian PowerPC systems, since those instructions do not work when the processor is in little endian mode. The exceptions are PPC740 and PPC750 which permit the instructions usage in little endian mode.

`-mupdate`
`-mno-update`
Generate code that uses (does not use) the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default. If you use '`-mno-update`', there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.

`-mfused-madd`
`-mno-fused-madd`
Generate code that uses (does not use) the floating point multiply and accumulate instructions. These instructions are generated by default if hardware floating is used.

`-mno-bit-align`
`-mbit-align`
: On System V.4 and embedded PowerPC systems do not (do) force structures and unions that contain bit fields to be aligned to the base type of the bit field.

  For example, by default a structure containing nothing but 8 `unsigned` bitfields of length 1 would be aligned to a 4 byte boundary and have a size of 4 bytes. By using '`-mno-bit-align`', the structure would be aligned to a 1 byte boundary and be one byte in size.

`-mno-strict-align`
`-mstrict-align`
: On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references will be handled by the system.

`-mrelocatable`
`-mno-relocatable`
: On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. If you use '`-mrelocatable`' on any module, all objects linked together must be compiled with '`-mrelocatable`' or '`-mrelocatable-lib`'.

`-mrelocatable-lib`
`-mno-relocatable-lib`
: On embedded PowerPC systems generate code that allows (does not allow) the program to be relocated to a different address at runtime. Modules compiled with '`-mrelocatable-lib`' can be linked with either modules compiled without '`-mrelocatable`' and '`-mrelocatable-lib`' or with modules compiled with the '`-mrelocatable`' options.

`-mno-toc`
`-mtoc`
: On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

`-mlittle`
`-mlittle-endian`
: On System V.4 and embedded PowerPC systems compile code for the processor in little endian mode. The '`-mlittle-endian`' option is the same as '`-mlittle`'.

`-mbig`
`-mbig-endian`
: On System V.4 and embedded PowerPC systems compile code for the processor in big endian mode. The '`-mbig-endian`' option is the same as '`-mbig`'.

`-mcall-sysv`
: On System V.4 and embedded PowerPC systems compile code using calling conventions that adheres to the March 1995 draft of the System V Application Binary Interface, PowerPC processor supplement. This is the default unless you configured GCC using '`powerpc-*-eabiaix`'.

`-mcall-sysv-eabi`
: Specify both '`-mcall-sysv`' and '`-meabi`' options.

`-mcall-sysv-noeabi`
: Specify both '`-mcall-sysv`' and '`-mno-eabi`' options.

`-mcall-aix`
: On System V.4 and embedded PowerPC systems compile code using calling conventions that are similar to those used on AIX. This is the default if you configured GCC using '`powerpc-*-eabiaix`'.

`-mcall-solaris`
: On System V.4 and embedded PowerPC systems compile code for the Solaris operating system.

`-mcall-linux`
: On System V.4 and embedded PowerPC systems compile code for the Linux-based GNU system.

`-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`'.

`-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.

2.14.13 IBM RT Options

These '`-m`' options are defined for the IBM RT PC:

`-min-line-mul`
: Use an in-line code sequence for integer multiplies. This is the default.

`-mcall-lib-mul`
: Call `lmul$$` for integer multiples.

`-mfull-fp-blocks`
: Generate full-size floating point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mminimum-fp-blocks`
: Do not include extra scratch space in floating point data blocks. This results in smaller code, but slower execution, since scratch space must be allocated dynamically.

`-mfp-arg-in-fpregs`
: Use a calling sequence incompatible with the IBM calling convention in which floating point arguments are passed in floating point registers. Note that `varargs.h` and `stdargs.h` will not work with floating point operands if this option is specified.

`-mfp-arg-in-gregs`
: Use the normal calling convention for floating point arguments. This is the default.

`-mhc-struct-return`
: Return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. Use the option '`-fpcc-struct-return`' for compatibility with the Portable C Compiler (pcc).

`-mnohc-struct-return`
: Return some structures of more than one word in registers, when convenient. This is the default. For compatibility with the IBM-supplied compilers, use the option '`-fpcc-struct-return`' or the option '`-mhc-struct-return`'.

2.14.14 MIPS Options

These '`-m`' options are defined for the MIPS family of computers:

`-mcpu=`*cpu type*
: Assume the defaults for the machine type *cpu type* when scheduling instructions. The choices for *cpu type* are '`r2000`', '`r3000`', '`r3900`', '`r4000`', '`r4100`', '`r4300`', '`r4400`', '`r4600`', '`r4650`', '`r5000`', '`r6000`', '`r8000`', and '`orion`'. Additionally, the '`r2000`', '`r3000`', '`r4000`', '`r5000`', and '`r6000`' can be abbreviated as '`r2k`' (or '`r2K`'), '`r3k`', etc. While picking a specific *cpu type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) without a '`-mipsX`' or '`-mabi`' switch being used.

`-mips1`
: Issue instructions from level 1 of the MIPS ISA. This is the default. '`r3000`' is the default *cpu type* at this ISA level.

`-mips2`
: Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). '`r6000`' is the default *cpu type* at this ISA level.

`-mips3`
: Issue instructions from level 3 of the MIPS ISA (64 bit instructions). '`r4000`' is the default *cpu type* at this ISA level.

`-mips4`
: Issue instructions from level 4 of the MIPS ISA (conditional move, prefetch, enhanced FPU instructions). '`r8000`' is the default *cpu type* at this ISA level.

`-mfp32`
: Assume that 32 32-bit floating point registers are available. This is the default.

`-mfp64`
: Assume that 32 64-bit floating point registers are available. This is the default when the '`-mips3`' option is used.

`-mgp32`
: Assume that 32 32-bit general purpose registers are available. This is the default.

`-mgp64`
: Assume that 32 64-bit general purpose registers are available. This is the default when the '`-mips3`' option is used.

`-mint64`
: Force int and long types to be 64 bits wide. See '`-mlong32`' for an explanation of the default, and the width of pointers.

`-mlong64`
: Force long types to be 64 bits wide. See '`-mlong32`' for an explanation of the default, and the width of pointers.

`-mlong32`
: Force long, int, and pointer types to be 32 bits wide.

  If none of '`-mlong32`', '`-mlong64`', or '`-mint64`' are set, the size of ints, longs, and pointers depends on the ABI and ISA choosen. For '`-mabi=32`', and '`-mabi=n32`', ints and longs are 32 bits wide. For '`-mabi=64`', ints are 32 bits, and longs are 64 bits wide. For '`-mabi=eabi`' and either '`-mips1`' or '`-mips2`', ints and longs are 32 bits wide. For '`-mabi=eabi`' and higher ISAs, ints are 32 bits, and longs are 64 bits wide. The width of pointer types is the smaller of the width of longs or the width of general purpose registers (which in turn depends on the ISA).

`-mabi=32`
`-mabi=o64`
`-mabi=n32`
`-mabi=64`
`-mabi=eabi`
: Generate code for the indicated ABI. The default instruction level is '`-mips1`' for '`32`', '`-mips3`' for '`n32`', and '`-mips4`' otherwise. Conversely, with '`-mips1`' or '`-mips2`', the default ABI is '`32`'; otherwise, the default ABI is '`64`'.

`-mmips-as`
: Generate code for the MIPS assembler, and invoke '`mips-tfile`' to add normal debug information. This is the default for all platforms except for the OSF/1 reference platform, using the OSF/rose object format. If the either of the '`-gstabs`' or '`-gstabs+`' switches are used, the '`mips-tfile`' program will encapsulate the stabs within MIPS ECOFF.

`-mgas`
: Generate code for the GNU assembler. This is the default on the OSF/1 reference platform, using the OSF/rose object format. Also, this is the default if the configure option '`--with-gnu-as`' is used.

`-msplit-addresses`
`-mno-split-addresses`

Generate code to load the high and low parts of address constants separately. This allows `gcc` to optimize away redundant loads of the high order bits of addresses. This optimization requires GNU as and GNU ld. This optimization is enabled by default for some embedded targets where GNU as and GNU ld are standard.

`-mrnames`
`-mno-rnames`

The '`-mrnames`' switch says to output code using the MIPS software names for the registers, instead of the hardware names (ie, *a0* instead of *$4*). The only known assembler that supports this option is the Algorithmics assembler.

`-mgpopt`
`-mno-gpopt`

The '`-mgpopt`' switch says to write all of the data declarations before the instructions in the text section, this allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

`-mstats`
`-mno-stats`

For each non-inline function processed, the '`-mstats`' switch causes the compiler to emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`
`-mno-memcpy`

The '`-mmemcpy`' switch makes all block moves call the appropriate string function ('`memcpy`' or '`bcopy`') instead of possibly generating inline code.

`-mmips-tfile`
`-mno-mips-tfile`

The '`-mno-mips-tfile`' switch causes the compiler not postprocess the object file with the '`mips-tfile`' program, after the MIPS assembler has generated it to add debug support. If '`mips-tfile`' is not run, then no local variables will be available to the debugger. In addition, '`stage2`' and '`stage3`' objects will have the temporary file names passed to the assembler embedded in the object file, which means the objects will not compare the same. The '`-mno-mips-tfile`' switch should only be used when there are bugs in the '`mips-tfile`' program that prevents compilation.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mhard-float`
: Generate output containing floating point instructions. This is the default if you use the unmodified sources.

`-mabicalls`
`-mno-abicalls`
: Emit (or do not emit) the pseudo operations '`.abicalls`', '`.cpload`', and '`.cprestore`' that some System V.4 ports use for position independent code.

`-mlong-calls`
`-mno-long-calls`
: Do all calls with the '`JALR`' instruction, which requires loading up a function's address into a register before the call. You need to use this switch, if you call outside of the current 512 megabyte segment to functions that are not through pointers.

`-mhalf-pic`
`-mno-half-pic`
: Put pointers to extern references into the data section and load them up, rather than put the references in the text section.

`-membedded-pic`
`-mno-embedded-pic`
: Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the $gp register. No more than 65536 bytes of global data may be used. This requires GNU as and GNU ld which do most of the work. This currently only works on targets which use ECOFF; it does not work with ELF.

`-membedded-data`
`-mno-embedded-data`
: Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-msingle-float`
`-mdouble-float`
: The '`-msingle-float`' switch tells gcc to assume that the floating point coprocessor only supports single precision operations, as on the '`r4650`' chip. The '`-mdouble-float`' switch permits gcc to use double precision operations. This is the default.

`-mmad`
`-mno-mad`
: Permit use of the '`mad`', '`madu`' and '`mul`' instructions, as on the '`r4650`' chip.

`-m4650`
: Turns on '`-msingle-float`', '`-mmad`', and, at least for now, '`-mcpu=r4650`'.

`-mips16`
`-mno-mips16`
: Enable 16-bit instructions.

`-mentry`
: Use the entry and exit pseudo ops. This option can only be used with '`-mips16`'.

`-EL` Compile code for the processor in little endian mode. The requisite libraries are assumed to exist.

`-EB` Compile code for the processor in big endian mode. The requisite libraries are assumed to exist.

`-G` *num* Put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (*gp* or *$28*), instead of the normal two words used. By default, *num* is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The '`-G` *num*' switch is also passed to the assembler and linker. All modules should be compiled with the same '`-G` *num*' value.

`-nocpp` Tell the MIPS assembler to not run its preprocessor over user assembler files (with a '`.s`' suffix) when assembling them.

2.14.15 Intel 386 Options

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

`-mcpu=`*cpu type*

Assume the defaults for the machine type *cpu type* when scheduling instructions. The choices for *cpu type* are:

'`i386`'	'`i486`'	'`i586`'	'`i686`'
'`pentium`'	'`pentiumpro`'	'`k6`'	

While picking a specific *cpu type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not run on the i386 without the '`-march=`*cpu type*' option being used. '`i586`' is equivalent to '`pentium`' and '`i686`' is equivalent to '`pentiumpro`'. '`k6`' is the AMD chip as opposed to the Intel ones.

`-march=`*cpu type*

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

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

Synonyms for -mcpu=i386, -mcpu=i486, -mcpu=pentium, and -mcpu=pentiumpro respectively. These synonyms are deprecated.

`-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. As of revision 2.6.1, these instructions are not generated unless you also use the '`-ffast-math`' switch.

`-malign-double`
`-mno-align-double`
: Control whether 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.

`-msvr3-shlib`
`-mno-svr3-shlib`
: Control whether GCC places uninitialized locals into `bss` or `data`. '`-msvr3-shlib`' places these locals into `bss`. These options are meaningful only on System V Release 3.

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

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

  You can specify that an individual function is called with this calling sequence with the function attribute '`stdcall`'. You can also override the '`-mrtd`' option by using the function attribute '`cdecl`'. See Section 4.23 [Function Attributes], page 151.

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

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

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

`-mreg-alloc=`*regs*

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

`-mregparm=`*num*

Control how many registers are used to pass integer arguments. By default, no registers are used to pass arguments, and at most 3 registers can be used. You can control this behavior for a specific function by using the function attribute '`regparm`'. See Section 4.23 [Function Attributes], page 151.

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

`-malign-loops=`*num*

Align loops to a 2 raised to a *num* byte boundary. If '`-malign-loops`' is not specified, the default is 2 unless gas 2.8 (or later) is being used in which case the default is to align the loop on a 16 byte boundary if it is less than 8 bytes away.

`-malign-jumps=`*num*

Align instructions that are only jumped to to a 2 raised to a *num* byte boundary. If '`-malign-jumps`' is not specified, the default is 2 if optimizing for a 386, and 4 if optimizing for a 486 unless gas 2.8 (or later) is being used in which case the default is to align the instruction on a 16 byte boundary if it is less than 8 bytes away.

`-malign-functions=`*num*

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

`-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).

The stack is required to be aligned on a 4 byte boundary. 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 Extention (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. 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`'.

2.14.16 HPPA Options

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

`-march=`*architecture type*
: Generate code for the specified architecture. The choices for *architecture type* are '`1.0`' for PA 1.0, '`1.1`' for PA 1.1, and '`2.0`' for PA 2.0 processors. Refer to '`/usr/lib/sched.models`' on an HP-UX system to determine the proper architecture option for your machine. Code compiled for lower numbered architectures will run on higher numbered architectures, but not the other way around.

  PA 2.0 support currently requires gas snapshot 19990413 or later. The next release of binutils (current is 2.9.1) will probably contain PA 2.0 support.

`-mpa-risc-1-0`
`-mpa-risc-1-1`
`-mpa-risc-2-0`
: Synonyms for -march=1.0, -march=1.1, and -march=2.0 respectively.

`-mbig-switch`
: Generate code suitable for big switch tables. Use this option only if the assembler/linker complain about out of range branches within a switch table.

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

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

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

`-mno-space-regs`
: Generate code that assumes the target has no space registers. This allows GCC to generate faster indirect calls and use unscaled index address modes.

  Such code is suitable for level 0 PA systems and kernels.

`-mfast-indirect-calls`
: Generate code that assumes calls never cross space boundaries. This allows GCC to emit code which performs faster indirect calls.

  This option will not work in the presense of shared libraries or nested functions.

`-mspace`
: Optimize for space rather than execution time. Currently this only enables out of line function prologues and epilogues. This option is incompatible with PIC code generation and profiling.

`-mlong-load-store`
: Generate 3-instruction load and store sequences as sometimes required by the HP-UX 10 linker. This is equivalent to the '`+k`' option to the HP compilers.

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

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

`-mschedule=`*cpu type*
: Schedule code according to the constraints for the machine type *cpu type*. The choices for *cpu type* are '`700`' '`7100`', '`7100LC`', '`7200`', and '`8000`'. Refer to '`/usr/lib/sched.models`' on an HP-UX system to determine the proper scheduling option for your machine.

`-mlinker-opt`
: Enable the optimization pass in the HPUX linker. Note this makes symbolic debugging impossible. It also triggers a bug in the HPUX 8 and HPUX 9 linkers in which they give bogus error messages when linking some programs.

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

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

2.14.17 Intel 960 Options

These '`-m`' options are defined for the Intel 960 implementations:

`-m`*cpu type*
: Assume the defaults for the machine type *cpu type* for some of the other options, including instruction scheduling, floating point support, and addressing modes. The choices for *cpu type* are '`ka`', '`kb`', '`mc`', '`ca`', '`cf`', '`sa`', and '`sb`'. The default is '`kb`'.

`-mnumerics`
`-msoft-float`
: The '`-mnumerics`' option indicates that the processor does support floating-point instructions. The '`-msoft-float`' option indicates that floating-point support should not be assumed.

`-mleaf-procedures`
`-mno-leaf-procedures`
: Do (or do not) attempt to alter leaf procedures to be callable with the `bal` instruction as well as `call`. This will result in more efficient code for explicit calls when the `bal` instruction can be substituted by the assembler or linker, but less efficient code in other cases, such as calls via function pointers, or using a linker that doesn't support this optimization.

`-mtail-call`
`-mno-tail-call`
: Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimize tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete. The default is '`-mno-tail-call`'.

`-mcomplex-addr`
`-mno-complex-addr`
: Assume (or do not assume) that the use of a complex addressing mode is a win on this implementation of the i960. Complex addressing modes may not be worthwhile on the K-series, but they definitely are on the C-series. The default is currently '`-mcomplex-addr`' for all processors except the CB and CC.

`-mcode-align`
`-mno-code-align`
: Align code to 8-byte boundaries for faster fetching (or don't bother). Currently turned on by default for C-series implementations only.

`-mic-compat`
`-mic2.0-compat`
`-mic3.0-compat`
: Enable compatibility with iC960 v2.0 or v3.0.

`-masm-compat`
`-mintel-asm`
: Enable compatibility with the iC960 assembler.

`-mstrict-align`
`-mno-strict-align`
: Do not permit (do permit) unaligned accesses.

`-mold-align`
: Enable structure-alignment compatibility with Intel's gcc release version 1.3 (based on gcc 1.37). This option implies '`-mstrict-align`'.

`-mlong-double-64`
: Implement type '`long double`' as 64-bit floating point numbers. Without the option '`long double`' is implemented by 80-bit floating point numbers. The only reason we have it because there is no 128-bit '`long double`' support in '`fp-bit.c`' yet. So it is only useful for people using soft-float targets. Otherwise, we should recommend against use of it.

2.14.18 DEC Alpha Options

These '`-m`' options are defined for the DEC Alpha implementations:

`-mno-soft-float`
`-msoft-float`
: Use (do not use) the hardware floating-point instructions for floating-point operations. When `-msoft-float` is specified, functions in '`libgcc1.c`' will be used to perform floating-point operations. Unless they are replaced by routines that emulate the floating-point operations, or compiled in such a way as to call such emulations routines, these routines will issue floating-point operations. If you are compiling for an Alpha without floating-point operations, you must ensure that the library is built so as not to call them.

  Note that Alpha implementations without floating-point operations are required to have floating-point registers.

`-mfp-reg`
`-mno-fp-regs`
: Generate code that uses (does not use) the floating-point register set. `-mno-fp-regs` implies `-msoft-float`. If the floating-point register set is not used, floating point operands are passed in integer registers as if they were integers and floating-point results are passed in $0 instead of $f0. This is a non-standard calling sequence, so any function with a floating-point argument or return value called by code compiled with `-mno-fp-regs` must also be compiled with that option.

  A typical use of this option is building a kernel that does not use, and hence need not save and restore, any floating-point registers.

`-mieee`
: The Alpha architecture implements floating-point hardware optimized for maximum performance. It is mostly compliant with the IEEE floating point standard. However, for full compliance, software assistance is required. This option generates code fully IEEE compliant code *except* that the *inexact flag* is not maintained (see below). If this option is turned on, the CPP macro `_IEEE_FP` is defined during compilation. The option is a shorthand for: '`-D_IEEE_FP -mfp-trap-mode=su -mtrap-precision=i -mieee-conformant`'. The resulting code is less efficient but is able to correctly support denormalized numbers and exceptional IEEE values such as not-a-number and plus/minus infinity. Other Alpha compilers call this option `-ieee_with_no_inexact`.

`-mieee-with-inexact`

This is like '`-mieee`' except the generated code also maintains the IEEE *inexact flag*. Turning on this option causes the generated code to implement fully-compliant IEEE math. The option is a shorthand for '`-D_IEEE_FP -D_IEEE_FP_INEXACT`' plus the three following: '`-mieee-conformant`', '`-mfp-trap-mode=sui`', and '`-mtrap-precision=i`'. On some Alpha implementations the resulting code may execute significantly slower than the code generated by default. Since there is very little code that depends on the *inexact flag*, you should normally not specify this option. Other Alpha compilers call this option '`-ieee_with_inexact`'.

`-mfp-trap-mode=`*trap mode*

This option controls what floating-point related traps are enabled. Other Alpha compilers call this option '`-fptm` '*trap mode*. The trap mode can be set to one of four values:

'`n`'	This is the default (normal) setting. The only traps that are enabled are the ones that cannot be disabled in software (e.g., division by zero trap).
'`u`'	In addition to the traps enabled by '`n`', underflow traps are enabled as well.
'`su`'	Like '`su`', but the instructions are marked to be safe for software completion (see Alpha architecture manual for details).
'`sui`'	Like '`su`', but inexact traps are enabled as well.

`-mfp-rounding-mode=`*rounding mode*

Selects the IEEE rounding mode. Other Alpha compilers call this option '`-fprm` '*rounding mode*. The *rounding mode* can be one of:

'`n`'	Normal IEEE rounding mode. Floating point numbers are rounded towards the nearest machine number or towards the even machine number in case of a tie.
'`m`'	Round towards minus infinity.
'`c`'	Chopped rounding mode. Floating point numbers are rounded towards zero.
'`d`'	Dynamic rounding mode. A field in the floating point control register (*fpcr*, see Alpha architecture reference manual) controls the rounding mode in effect. The C library initializes this register for rounding towards plus infinity. Thus, unless your program modifies the *fpcr*, '`d`' corresponds to round towards plus infinity.

`-mtrap-precision=`*trap precision*

In the Alpha architecture, floating point traps are imprecise. This means without software assistance it is impossible to recover from a floating trap and program execution normally needs to be terminated. GCC can generate code that can assist operating system trap handlers in determining the exact location that caused a floating point trap. Depending on the requirements of an application, different levels of precisions can be selected:

‘p’ Program precision. This option is the default and means a trap handler can only identify which program caused a floating point exception.

‘f’ Function precision. The trap handler can determine the function that caused a floating point exception.

‘i’ Instruction precision. The trap handler can determine the exact instruction that caused a floating point exception.

Other Alpha compilers provide the equivalent options called ‘`-scope_safe`’ and ‘`-resumption_safe`’.

`-mieee-conformant`
This option marks the generated code as IEEE conformant. You must not use this option unless you also specify ‘`-mtrap-precision=i`’ and either ‘`-mfp-trap-mode=su`’ or ‘`-mfp-trap-mode=sui`’. Its only effect is to emit the line ‘`.eflag 48`’ in the function prologue of the generated assembly file. Under DEC Unix, this has the effect that IEEE-conformant math library routines will be linked in.

`-mbuild-constants`
Normally GCC examines a 32- or 64-bit integer constant to see if it can construct it from smaller constants in two or three instructions. If it cannot, it will output the constant as a literal and generate code to load it from the data segment at runtime.

Use this option to require GCC to construct *all* integer constants using code, even if it takes more instructions (the maximum is six).

You would typically use this option to build a shared library dynamic loader. Itself a shared library, it must relocate itself in memory before it can find the variables and constants in its own data segment.

`-malpha-as`
`-mgas` Select whether to generate code to be assembled by the vendor-supplied assembler (‘`-malpha-as`’) or by the GNU assembler ‘`-mgas`’.

`-mbwx`
`-mno-bwx`
`-mcix`
`-mno-cix`
`-mmax`
`-mno-max` Indicate whether GCC should generate code to use the optional BWX, CIX, and MAX instruction sets. The default is to use the instruction sets supported by the CPU type specified via ‘`-mcpu=`’ option or that of the CPU on which GCC was built if none was specified.

`-mcpu=`*cpu_type*
Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. You can specify either the ‘`EV`’ style name or the corresponding chip number. GCC supports scheduling parameters for the EV4 and EV5 family of processors and will choose the default values for the instruction

set from the processor you specify. If you do not specify a processor type, GCC will default to the processor on which the compiler was built.

Supported values for *cpu_type* are

`ev4`
`21064` Schedules as an EV4 and has no instruction set extensions.

`ev5`
`21164` Schedules as an EV5 and has no instruction set extensions.

`ev56`
`21164a` Schedules as an EV5 and supports the BWX extension.

`pca56`
`21164pc`
`21164PC` Schedules as an EV5 and supports the BWX and MAX extensions.

`ev6`
`21264` Schedules as an EV5 (until Digital releases the scheduling parameters for the EV6) and supports the BWX, CIX, and MAX extensions.

`-mmemory-latency=`*time*
Sets the latency the scheduler should assume for typical memory references as seen by the application. This number is highly dependant on the memory access patterns used by the application and the size of the external cache on the machine.

Valid options for *time* are

'*number*' A decimal number representing clock cycles.

`L1`
`L2`
`L3`
`main` The compiler contains estimates of the number of clock cycles for "typical" EV4 & EV5 hardware for the Level 1, 2 & 3 caches (also called Dcache, Scache, and Bcache), as well as to main memory. Note that L3 is only valid for EV5.

2.14.19 Clipper Options

These '`-m`' options are defined for the Clipper implementations:

`-mc300` Produce code for a C300 Clipper processor. This is the default.

`-mc400` Produce code for a C400 Clipper processor i.e. use floating point registers f8..f15.

2.14.20 H8/300 Options

These '`-m`' options are defined for the H8/300 implementations:

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '`-relax`'. See section "`ld` and the H8/300" in *Using ld*, for a fuller description.

`-mh` Generate code for the H8/300H.

`-ms` Generate code for the H8/S.

`-mint32` Make `int` data 32 bits by default.

`-malign-300`
On the h8/300h, use the same alignment rules as for the h8/300. The default for the h8/300h is to align longs and floats on 4 byte boundaries. '`-malign-300`' causes them to be aligned on 2 byte boundaries. This option has no effect on the h8/300.

2.14.21 SH Options

These '`-m`' options are defined for the SH implementations:

`-m1` Generate code for the SH1.

`-m2` Generate code for the SH2.

`-m3` Generate code for the SH3.

`-m3e` Generate code for the SH3e.

`-mb` Compile code for the processor in big endian mode.

`-ml` Compile code for the processor in little endian mode.

`-mdalign` Align doubles at 64 bit boundaries. Note that this changes the calling conventions, and thus some functions from the standard C library will not work unless you recompile it first with -mdalign.

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '`-relax`'.

2.14.22 Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

`-G` Create a shared object. It is recommended that '`-symbolic`' or '`-shared`' be used instead.

`-Qy` Identify the versions of each tool used by the compiler, in a `.ident` assembler directive in the output.

`-Qn` Refrain from adding `.ident` directives to the output file (this is the default).

`-YP,`*dirs* Search the directories *dirs*, and no others, for libraries specified with '`-l`'.

`-Ym,`*dir* Look in the directory *dir* to find the M4 preprocessor. The assembler uses this option.

2.14.23 TMS320C3x/C4x Options

These '`-m`' options are defined for TMS320C3x/C4x implementations:

`-mcpu=`*cpu_type*
: Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`c30`', '`c31`', '`c32`', '`c40`', and '`c44`'. The default is '`c40`' to generate code for the TMS320C40.

`-mbig-memory`
`-mbig`
`-msmall-memory`
`-msmall`
: Generates code for the big or small memory model. The small memory model assumed that all data fits into one 64K word page. At run-time the data page (DP) register must be set to point to the 64K page containing the .bss and .data program sections. The big memory model is the default and requires reloading of the DP register for every direct memory access.

`-mbk`
`-mno-bk`
: Allow (disallow) allocation of general integer operands into the block count register BK.

`-mdb`
`-mno-db`
: Enable (disable) generation of code using decrement and branch, DBcond(D), instructions. This is enabled by default for the C4x. To be on the safe side, this is disabled for the C3x, since the maximum iteration count on the C3x is 2^23 + 1 (but who iterates loops more than 2^23 times on the C3x?). Note that GCC will try to reverse a loop so that it can utilise the decrement and branch instruction, but will give up if there is more than one memory reference in the loop. Thus a loop where the loop counter is decremented can generate slightly more efficient code, in cases where the RPTB instruction cannot be utilised.

`-mdp-isr-reload`
`-mparanoid`
: Force the DP register to be saved on entry to an interrupt service routine (ISR), reloaded to point to the data section, and restored on exit from the ISR. This should not be required unless someone has violated the small memory model by modifying the DP register, say within an object library.

`-mmpyi`
`-mno-mpyi`
: For the C3x use the 24-bit MPYI instruction for integer multiplies instead of a library call to guarantee 32-bit results. Note that if one of the operands is a constant, then the multiplication will be performed using shifts and adds. If the -mmpyi option is not specified for the C3x, then squaring operations are performed inline instead of a library call.

`-mfast-fix`
`-mno-fast-fix`
: The C3x/C4x FIX instruction to convert a floating point value to an integer value chooses the nearest integer less than or equal to the floating point value

rather than to the nearest integer. Thus if the floating point number is negative, the result will be incorrectly truncated an additional code is necessary to detect and correct this case. This option can be used to disable generation of the additional code required to correct the result.

`-mrptb`
`-mno-rptb`
: Enable (disable) generation of repeat block sequences using the RPTB instruction for zero overhead looping. The RPTB construct is only used for innermost loops that do not call functions or jump across the loop boundaries. There is no advantage having nested RPTB loops due to the overhead required to save and restore the RC, RS, and RE registers. This is enabled by default with -O2.

`-mrpts=`*count*
`-mno-rpts`
: Enable (disable) the use of the single instruction repeat instruction RPTS. If a repeat block contains a single instruction, and the loop count can be guaranteed to be less than the value *count*, GCC will emit a RPTS instruction instead of a RPTB. If no value is specified, then a RPTS will be emitted even if the loop count cannot be determined at compile time. Note that the repeated instruction following RPTS does not have to be reloaded from memory each iteration, thus freeing up the CPU buses for oeprands. However, since interrupts are blocked by this instruction, it is disabled by default.

`-mloop-unsigned`
`-mno-loop-unsigned`
: The maximum iteration count when using RPTS and RPTB (and DB on the C40) is 2^31 + 1 since these instructions test if the iteration count is negative to terminate the loop. If the iteration count is unsigned there is a possibility than the 2^31 + 1 maximum iteration count may be exceeded. This switch allows an unsigned iteration count.

`-mti`
: Try to emit an assembler syntax that the TI assembler (asm30) is happy with. This also enforces compatibility with the API employed by the TI C3x C compiler. For example, long doubles are passed as structures rather than in floating point registers.

`-mregparm`
`-mmemparm`
: Generate code that uses registers (stack) for passing arguments to functions. By default, arguments are passed in registers where possible rather than by pushing arguments on to the stack.

`-mparallel-insns`
`-mno-parallel-insns`
: Allow the generation of parallel instructions. This is enabled by default with -O2.

`-mparallel-mpy`
`-mno-parallel-mpy`
: Allow the generation of MPY||ADD and MPY||SUB parallel instructions, provided -mparallel-insns is also specified. These instructions have tight register constraints which can pessimize the code generation of large functions.

2.14.24 V850 Options

These '`-m`' options are defined for V850 implementations:

`-mlong-calls`
`-mno-long-calls`
: Treat all calls as being far away (near). If calls are assumed to be far away, the compiler will always load the functions address up into a register, and call indirect through the pointer.

`-mno-ep`
`-mep`
: Do not optimize (do optimize) basic blocks that use the same index pointer 4 or more times to copy pointer into the `ep` register, and use the shorter `sld` and `sst` instructions. The '`-mep`' option is on by default if you optimize.

`-mno-prolog-function`
`-mprolog-function`
: Do not use (do use) external functions to save and restore registers at the prolog and epilog of a function. The external functions are slower, but use less code space if more than one function saves the same number of registers. The '`-mprolog-function`' option is on by default if you optimize.

`-mspace`
: Try to make the code as small as possible. At present, this just turns on the '`-mep`' and '`-mprolog-function`' options.

`-mtda=`*n*
: Put static or global variables whose size is *n* bytes or less into the tiny data area that register `ep` points to. The tiny data area can hold up to 256 bytes in total (128 bytes for byte references).

`-msda=`*n*
: Put static or global variables whose size is *n* bytes or less into the small data area that register `gp` points to. The small data area can hold up to 64 kilobytes.

`-mzda=`*n*
: Put static or global variables whose size is *n* bytes or less into the first 32 kilobytes of memory.

`-mv850`
: Specify that the target processor is the V850.

`-mbig-switch`
: Generate code suitable for big switch tables. Use this option only if the assembler/linker complain about out of range branches within a switch table.

2.14.25 ARC Options

These options are defined for ARC implementations:

`-EL`
: Compile code for little endian mode. This is the default.

`-EB`
: Compile code for big endian mode.

`-mmangle-cpu`
: Prepend the name of the cpu to all public symbol names. In multiple-processor systems, there are many ARC variants with different instruction and register set characteristics. This flag prevents code compiled for one cpu to be linked with code compiled for another. No facility exists for handling variants that are "almost identical". This is an all or nothing option.

`-mcpu=`*cpu*
: Compile code for ARC variant *cpu*. Which variants are supported depend on the configuration. All variants support '`-mcpu=base`', this is the default.

`-mtext=`*text section*
`-mdata=`*data section*
`-mrodata=`*readonly data section*
: Put functions, data, and readonly data in *text section*, *data section*, and *readonly data section* respectively by default. This can be overridden with the `section` attribute. See Section 4.29 [Variable Attributes], page 158.

2.14.26 NS32K Options

These are the '`-m`' options defined for the 32000 series. The default values for these options depends on which style of 32000 was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m32032`
`-m32032` Generate output for a 32032. This is the default when the compiler is configured for 32032 and 32016 based systems.

`-m32332`
`-m32332` Generate output for a 32332. This is the default when the compiler is configured for 32332-based systems.

`-m32532`
`-m32532` Generate output for a 32532. This is the default when the compiler is configured for 32532-based systems.

`-m32081` Generate output containing 32081 instructions for floating point. This is the default for all systems.

`-m32381` Generate output containing 32381 instructions for floating point. This also implies '`-m32081`'. The 32381 is only compatible with the 32332 and 32532 cpus. This is the default for the pc532-netbsd configuration.

`-mmulti-add`
: Try and generate multiply-add floating point instructions `polyF` and `dotF`. This option is only available if the '`-m32381`' option is in effect. Using these instructions requires changes to to register allocation which generally has a negative impact on performance. This option should only be enabled when compiling code particularly likely to make heavy use of multiply-add instructions.

`-mnomulti-add`
: Do not try and generate multiply-add floating point instructions `polyF` and `dotF`. This is the default on all platforms.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries may not be available.

`-mnobitfield`
: Do not use the bit-field instructions. On some machines it is faster to use shifting and masking operations. This is the default for the pc532.

`-mbitfield`
: Do use the bit-field instructions. This is the default for all platforms except the pc532.

`-mrtd`
: Use a different function-calling convention, in which functions that take a fixed number of arguments return pop their arguments on return with the `ret` instruction.

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

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

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

  This option takes its name from the 680x0 `rtd` instruction.

`-mregparam`
: Use a different function-calling convention where the first two arguments are passed in registers.

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

`-mnoregparam`
: Do not pass any arguments in registers. This is the default for all targets.

`-msb`
: It is OK to use the sb as an index register which is always loaded with zero. This is the default for the pc532-netbsd target.

`-mnosb`
: The sb register is not available for use or has not been initialized to zero by the run time system. This is the default for all targets except the pc532-netbsd. It is also implied whenever '`-mhimem`' or '`-fpic`' is set.

`-mhimem`
: Many ns32000 series addressing modes use displacements of up to 512MB. If an address is above 512MB then displacements from zero can not be used. This option causes code to be generated which can be loaded above 512MB. This may be useful for operating systems or ROM code.

`-mnohimem`
: Assume code will be loaded in the first 512MB of virtual address space. This is the default for all platforms.

2.15 Options for Code Generation Conventions

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

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

`-fexceptions`
: Enable exception handling. Generates extra code needed to propagate exceptions. For some targets, this implies generation of frame unwind information for all functions. This can produce significant data size overhead, although it does not affect execution. If you do not specify this option, it is enabled by default for languages like C++ which normally require exception handling, and disabled for languages like C that do not normally require it. However, when compiling C code that needs to interoperate properly with exception handlers written in C++, you may need to enable this option. You may also wish to disable this option is you are compiling older C++ programs that don't use exception handling.

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

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

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

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

  If you specify neither '`-fpcc-struct-return`' nor its contrary '`-freg-struct-return`', GCC defaults to whichever convention is standard for the target. If there is no standard convention, GCC defaults to '`-fpcc-struct-return`', except on targets where GCC is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

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

`-fshort-double`
: Use the same size for `double` as for `float`.

`-fshared-data`
: Requests that the data and non-`const` variables of this compilation be shared data rather than private data. The distinction makes sense only on certain

operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

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

`-fno-ident`
: Ignore the '`#ident`' directive.

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

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

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

  '`-fno-verbose-asm`', the default, causes the extra information to be omitted and is useful when comparing two assembler files.

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

`-fvolatile-global`
: Consider all memory references to extern and global data items to be volatile. GCC does not consider static data items to be volatile because of this switch.

`-fvolatile-static`
: Consider all memory references to static data to be volatile.

`-fpic`
: Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of GCC; it is part of the operating system). If the GOT size for the linked executable

exceeds a machine-specific maximum size, you get an error message from the linker indicating that '`-fpic`' does not work; in that case, recompile with '`-fPIC`' instead. (These maximums are 16k on the m88k, 8k on the Sparc, and 32k on the m68k and RS/6000. The 386 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines. For the 386, GCC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

`-fPIC` If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, m88k, and the Sparc.

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

`-ffixed-`*reg*

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

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

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

`-fcall-used-`*reg*

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

It is an error to used this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results.

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

`-fcall-saved-`*reg*

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

It is an error to used this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results.

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

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

`-fpack-struct`

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

`-fcheck-memory-usage`

Generate extra code to check each memory access. GCC will generate code that is suitable for a detector of bad memory accesses such as '`Checker`'.

Normally, you should compile all, or none, of your code with this option.

If you do mix code compiled with and without this option, you must ensure that all code that has side effects and that is called by code compiled with this option is, itself, compiled with this option. If you do not, you might get erroneous messages from the detector.

If you use functions from a library that have side-effects (such as `read`), you might not be able to recompile the library and specify this option. In that case, you can enable the '`-fprefix-function-name`' option, which requests GCC to encapsulate your code and make other functions look as if they were compiled with '`-fcheck-memory-usage`'. This is done by calling "stubs", which are provided by the detector. If you cannot find or build stubs for every function you call, you might have to specify '`-fcheck-memory-usage`' without '`-fprefix-function-name`'.

If you specify this option, you can not use the `asm` or `__asm__` keywords in functions with memory checking enabled. The compiler cannot understand what the `asm` statement will do, and therefore cannot generate the appropriate code, so it is rejected. However, the function attribute `no_check_memory_usage` will disable memory checking within a function, and `asm` statements can be put inside such functions. Inline expansion of a non-checked function within a checked function is permitted; the inline function's memory accesses won't be checked, but the rest will.

If you move your `asm` statements to non-checked inline functions, but they do access memory, you can add calls to the support code in your inline function, to indicate any reads, writes, or copies being done. These calls would be similar to those done in the stubs described above.

`-fprefix-function-name`

Request GCC to add a prefix to the symbols generated for function names. GCC adds a prefix to the names of functions defined as well as functions called. Code compiled with this option and code compiled without the option can't be linked together, unless stubs are used.

If you compile the following code with '`-fprefix-function-name`'

```
extern void bar (int);
void
foo (int a)
{
  return bar (a + 5);
}
```

GCC will compile the code as if it was written:

```
extern void prefix_bar (int);
void
prefix_foo (int a)
{
```

```
            return prefix_bar (a + 5);
          }
```

This option is designed to be used with '`-fcheck-memory-usage`'.

`-finstrument-functions`

Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site. (On some platforms, `__builtin_return_address` does not work beyond the current function, so the call site information may not be available to the profiling functions otherwise.)

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table.

This instrumentation is also done for functions expanded inline in other functions. The profiling calls will indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use '`extern inline`' in your C code, an addressable version of such functions must be provided. (This is normally the case anyways, but if you get lucky and the optimizer always expands the functions inline, you might have gotten away without providing static copies.)

A function may be given the attribute `no_instrument_function`, in which case this instrumentation will not be done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory).

`-fstack-check`

Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

`-fargument-alias`
`-fargument-noalias`
`-fargument-noalias-global`

Specify the possible relationships among parameters and between parameters and global data.

'`-fargument-alias`' specifies that arguments (parameters) may alias each other and may alias global storage. '`-fargument-noalias`' specifies that arguments do not alias each other, but may alias global storage. '`-fargument-noalias-global`' specifies that arguments do not alias each other and do not alias global storage.

Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.

`-fleading-underscore`

This option and its counterpart, -fno-leading-underscore, forcibly change the way C symbols are represented in the object file. One use is to help link with legacy assembly code.

Be warned that you should know what you are doing when invoking this option, and that not all targets provide complete support for it.

2.16 Environment Variables Affecting GCC

This section describes several environment variables that affect how GCC operates. Some of them work by specifying directories or prefixes to use when searching for various kinds of files. Some are used to specify other aspects of the compilation environment.

Note that you can also specify places to search using options such as '`-B`', '`-I`' and '`-L`' (see Section 2.12 [Directory Options], page 46). 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_UK`' for English in the United Kingdom.

The `LC_CTYPE` environment variable specifies character classification. GCC uses it to determine the character boundaries in a string; this is needed for some multibyte encodings that contain quote and escape characters that would otherwise be interpreted as a string end or escape.

The `LC_MESSAGES` environment variable specifies the language to use in diagnostic messages.

If the `LC_ALL` environment variable is set, it overrides the value of `LC_CTYPE` and `LC_MESSAGES`; otherwise, `LC_CTYPE` and `LC_MESSAGES` default to the value of the `LANG` environment variable. If none of these variables are set, GCC defaults to traditional C English behavior.

`TMPDIR`

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

`GCC_EXEC_PREFIX`

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

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

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

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

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

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

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

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

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

`DEPENDENCIES_OUTPUT`
: If this variable is set, its value specifies how to output dependencies for Make based on the header files processed by the compiler. This output looks much like the output from the '`-M`' option (see Section 2.9 [Preprocessor Options], page 41), but it goes to a separate file, and is in addition to the usual results of compilation.

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

`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.

2.17 Running Protoize

The program `protoize` is an optional part of GNU C. You can use it to add prototypes to a program, thus converting the program to ANSI 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-ANSI 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`'. If the '`.save`' 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`' 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
protoize *.c
```

You need to include the special files along with the rest in the protoize command, even though their '.X' files already exist, because otherwise they won't get converted.

See Section 7.11 [Protoize Caveats], page 211, for more information on how to use protoize successfully.

Note most of this information is out of date and superceded by the EGCS install procedures. It is provided for historical reference only.

3 Installing GNU CC

Here is the procedure for installing GNU CC on a GNU or Unix system. See Section 3.6 [VMS Install], page 131, for VMS systems. In this section we assume you compile in the same directory that contains the source files; see Section 3.3 [Other Dir], page 126, to find out how to compile in a separate directory on Unix systems.

You cannot install GNU C by itself on MSDOS; it will not compile under any MSDOS compiler except itself. You need to get the complete compilation package DJGPP, which includes binaries as well as sources, and includes all the necessary compilation tools and libraries.

1. If you have built GNU CC previously in the same directory for a different target machine, do '`make distclean`' to delete all files that might be invalid. One of the files this deletes is '`Makefile`'; if '`make distclean`' complains that '`Makefile`' does not exist, it probably means that the directory is already suitably clean.
2. On a System V release 4 system, make sure '`/usr/bin`' precedes '`/usr/ucb`' in `PATH`. The `cc` command in '`/usr/ucb`' uses libraries which have bugs.
3. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output files '`c-parse.c`' and '`cexp.c`' are more recent than '`c-parse.y`' and '`cexp.y`' and you do not plan to change the '`.y`' files.)

   Bison versions older than Sept 8, 1988 will produce incorrect output for '`c-parse.c`'.
4. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the build directory under the names '`as`', '`ld`' or whatever is appropriate. This will enable the compiler to find the proper tools for compilation of the program '`enquire`'.

   Alternatively, you can do subsequent compilation using a value of the `PATH` environment variable such that the necessary GNU tools come before the standard system tools.
5. Specify the host, build and target machine configurations. You do this when you run the '`configure`' script.

   The *build* machine is the system which you are using, the *host* machine is the system where you want to run the resulting compiler (normally the build machine), and the *target* machine is the system for which you want the compiler to generate code.

   If you are building a compiler to produce code for the machine it runs on (a native compiler), you normally do not need to specify any operands to '`configure`'; it will try to guess the type of machine you are on and use that as the build, host and target machines. So you don't need to specify a configuration when building a native compiler unless '`configure`' cannot figure out what your configuration is or guesses wrong.

   In those cases, specify the build machine's *configuration name* with the '`--host`' option; the host and target will default to be the same as the host machine. (If you are building a cross-compiler, see Section 3.4 [Cross-Compiler], page 126.)

   Here is an example:

   ```
   ./configure --host=sparc-sun-sunos4.1
   ```

   A configuration name may be canonical or it may be more or less abbreviated.

A canonical configuration name has three parts, separated by dashes. It looks like this: '*cpu-company-system*'. (The three parts may themselves contain dashes; '`configure`' can figure out which dashes serve which purpose.) For example, '`m68k-sun-sunos4.1`' specifies a Sun 3.

You can also replace parts of the configuration by nicknames or aliases. For example, '`sun3`' stands for '`m68k-sun`', so '`sun3-sunos4.1`' is another way to specify a Sun 3. You can also use simply '`sun3-sunos`', since the version of SunOS is assumed by default to be version 4.

You can specify a version number after any of the system types, and some of the CPU types. In most cases, the version is irrelevant, and will be ignored. So you might as well specify the version if you know it.

See Section 3.2 [Configurations], page 111, for a list of supported configuration names and notes on many of the configurations. You should check the notes in that section before proceeding any further with the installation of GNU CC.

6. When running `configure`, you may also need to specify certain additional options that describe variant hardware and software configurations. These are '`--with-gnu-as`', '`--with-gnu-ld`', '`--with-stabs`' and '`--nfp`'.

'`--with-gnu-as`'

If you will use GNU CC with the GNU assembler (GAS), you should declare this by using the '`--with-gnu-as`' option when you run '`configure`'.

Using this option does not install GAS. It only modifies the output of GNU CC to work with GAS. Building and installing GAS is up to you.

Conversely, if you *do not* wish to use GAS and do not specify '`--with-gnu-as`' when building GNU CC, it is up to you to make sure that GAS is not installed. GNU CC searches for a program named `as` in various directories; if the program it finds is GAS, then it runs GAS. If you are not sure where GNU CC finds the assembler it is using, try specifying '`-v`' when you run it.

The systems where it makes a difference whether you use GAS are '`hppa1.0-`*any*`-`*any*', '`hppa1.1-`*any*`-`*any*', '`i386-`*any*`-sysv`', '`i386-`*any*`-isc`',

'`i860-`*any*`-bsd`', '`m68k-bull-sysv`',
'`m68k-hp-hpux`', '`m68k-sony-bsd`',
'`m68k-altos-sysv`', '`m68000-hp-hpux`',
'`m68000-att-sysv`', '*any*`-lynx-lynxos`', and '`mips-`*any*'). On any other system, '`--with-gnu-as`' has no effect.

On the systems listed above (except for the HP-PA, for ISC on the 386, and for '`mips-sgi-irix5.*`'), if you use GAS, you should also use the GNU linker (and specify '`--with-gnu-ld`').

'`--with-gnu-ld`'

Specify the option '`--with-gnu-ld`' if you plan to use the GNU linker with GNU CC.

This option does not cause the GNU linker to be installed; it just modifies the behavior of GNU CC to work with the GNU linker.

'`--with-stabs`'
: On MIPS based systems and on Alphas, you must specify whether you want GNU CC to create the normal ECOFF debugging format, or to use BSD-style stabs passed through the ECOFF symbol table. The normal ECOFF debug format cannot fully handle languages other than C. BSD stabs format can handle other languages, but it only works with the GNU debugger GDB.

  Normally, GNU CC uses the ECOFF debugging format by default; if you prefer BSD stabs, specify '`--with-stabs`' when you configure GNU CC.

  No matter which default you choose when you configure GNU CC, the user can use the '`-gcoff`' and '`-gstabs+`' options to specify explicitly the debug format for a particular compilation.

  '`--with-stabs`' is meaningful on the ISC system on the 386, also, if '`--with-gas`' is used. It selects use of stabs debugging information embedded in COFF output. This kind of debugging information supports C++ well; ordinary COFF debugging information does not.

  '`--with-stabs`' is also meaningful on 386 systems running SVR4. It selects use of stabs debugging information embedded in ELF output. The C++ compiler currently (2.6.0) does not support the DWARF debugging information normally used on 386 SVR4 platforms; stabs provide a workable alternative. This requires gas and gdb, as the normal SVR4 tools can not generate or interpret stabs.

'`--nfp`'
: On certain systems, you must specify whether the machine has a floating point unit. These systems include '`m68k-sun-sunos`*n*' and '`m68k-isi-bsd`'. On any other system, '`--nfp`' currently has no effect, though perhaps there are other systems where it could usefully make a difference.

'`--enable-haifa`'
'`--disable-haifa`'
: Use '`--enable-haifa`' to enable use of an experimental instruction scheduler (from IBM Haifa). This may or may not produce better code. Some targets on which it is known to be a win enable it by default; use '`--disable-haifa`' to disable it in these cases. `configure` will print out whether the Haifa scheduler is enabled when it is run.

'`--enable-threads=`*type*'
: Certain systems, notably Linux-based GNU systems, can't be relied on to supply a threads facility for the Objective C runtime and so will default to single-threaded runtime. They may, however, have a library threads implementation available, in which case threads can be enabled with this option by supplying a suitable *type*, probably '`posix`'. The possibilities for *type* are '`single`', '`posix`', '`win32`', '`solaris`', '`irix`' and '`mach`'.

'`--enable-checking`'
: When you specify this option, the compiler is built to perform checking of tree node types when referencing fields of that node. This does not change the generated code, but adds error checking within the compiler. This will

slow down the compiler and may only work properly if you are building the compiler with GNU C.

The '`configure`' script searches subdirectories of the source directory for other compilers that are to be integrated into GNU CC. The GNU compiler for C++, called G++ is in a subdirectory named '`cp`'. '`configure`' inserts rules into '`Makefile`' to build all of those compilers.

Here we spell out what files will be set up by `configure`. Normally you need not be concerned with these files.

- A file named '`config.h`' is created that contains a '`#include`' of the top-level config file for the machine you will run the compiler on (see section "The Configuration File" in *Using and Porting GCC*). This file is responsible for defining information about the host machine. It includes '`tm.h`'.

  The top-level config file is located in the subdirectory '`config`'. Its name is always '`xm-`*something*`.h`'; usually '`xm-`*machine*`.h`', but there are some exceptions.

  If your system does not support symbolic links, you might want to set up '`config.h`' to contain a '`#include`' command which refers to the appropriate file.
- A file named '`tconfig.h`' is created which includes the top-level config file for your target machine. This is used for compiling certain programs to run on that machine.
- A file named '`tm.h`' is created which includes the machine-description macro file for your target machine. It should be in the subdirectory '`config`' and its name is often '*machine*`.h`'.

'`--enable-nls`'
'`--disable-nls`'
The '`--enable-nls`' option enables Native Language Support (NLS), which lets GCC output diagnostics in languages other than American English. No translations are available yet, so the main users of this option now are those translating GCC's diagnostics who want to test their work. Once translations become available, Native Language Support will become enabled by default. The '`--disable-nls`' option disables NLS.

'`--with-included-gettext`'
If NLS is enabled, the GCC build procedure normally attempts to use the host's `gettext` libraries, and falls back on GCC's copy of the GNU `gettext` library only if the host libraries do not suffice. The '`--with-included-gettext`' option causes the build procedure to prefer its copy of GNU `gettext`.

'`--with-catgets`'
If NLS is enabled, and if the host lacks `gettext` but has the inferior `catgets` interface, the GCC build procedure normally ignores `catgets` and instead uses GCC's copy of the GNU `gettext` library. The '`--with-catgets`' option causes the build procedure to use the host's `catgets` in this situation.

7. In certain cases, you should specify certain other options when you run `configure`.
   - The standard directory for installing GNU CC is '`/usr/local/lib`'. If you want to install its files somewhere else, specify '`--prefix=`*dir*' when you run '`configure`'. Here *dir* is a directory name to use instead of '`/usr/local`' for all purposes with one exception: the directory '`/usr/local/include`' is searched for header files no matter where you install the compiler. To override this name, use the `--with-local-prefix` option below. The directory you specify need not exist, but its parent directory must exist.
   - Specify '`--with-local-prefix=`*dir*' if you want the compiler to search directory '*dir*`/include`' for locally installed header files *instead* of '`/usr/local/include`'.

     You should specify '`--with-local-prefix`' **only** if your site has a different convention (not '`/usr/local`') for where to put site-specific files.

     The default value for '`--with-local-prefix`' is '`/usr/local`' regardless of the value of '`--prefix`'. Specifying '`--prefix`' has no effect on which directory GNU CC searches for local header files. This may seem counterintuitive, but actually it is logical.

     The purpose of '`--prefix`' is to specify where to *install GNU CC*. The local header files in '`/usr/local/include`'—if you put any in that directory—are not part of GNU CC. They are part of other programs—perhaps many others. (GNU CC installs its own header files in another directory which is based on the '`--prefix`' value.)

     Do not specify '`/usr`' as the '`--with-local-prefix`'! The directory you use for '`--with-local-prefix`' **must not** contain any of the system's standard header files. If it did contain them, certain programs would be miscompiled (including GNU Emacs, on certain targets), because this would override and nullify the header file corrections made by the `fixincludes` script.

     Indications are that people who use this option use it based on mistaken ideas of what it is for. People use it as if it specified where to install part of GNU CC. Perhaps they make this assumption because installing GNU CC creates the directory.
8. Build the compiler. Just type '`make LANGUAGES=c`' in the compiler directory.

   '`LANGUAGES=c`' specifies that only the C compiler should be compiled. The makefile normally builds compilers for all the supported languages; currently, C, C++ and Objective C. However, C is the only language that is sure to work when you build with other non-GNU C compilers. In addition, building anything but C at this stage is a waste of time.

   In general, you can specify the languages to build by typing the argument '`LANGUAGES="`*list*`"`', where *list* is one or more words from the list '`c`', '`c++`', and '`objective-c`'. If you have any additional GNU compilers as subdirectories of the GNU CC source directory, you may also specify their names in this list.

   Ignore any warnings you may see about "statement not reached" in '`insn-emit.c`'; they are normal. Also, warnings about "unknown escape sequence" are normal in '`genopinit.c`' and perhaps some other files. Likewise, you should ignore warnings about "constant is so large that it is unsigned" in '`insn-emit.c`' and '`insn-recog.c`', a warning about a comparison always being zero in '`enquire.o`', and warnings about shift counts exceeding type widths in '`cexp.y`'. Any other compilation errors may represent bugs in the port to your machine or operating system, and should be investigated and reported (see Chapter 8 [Bugs], page 217).

Some compilers fail to compile GNU CC because they have bugs or limitations. For example, the Microsoft compiler is said to run out of macro space. Some Ultrix compilers run out of expression space; then you need to break up the statement where the problem happens.

9. If you are building a cross-compiler, stop here. See Section 3.4 [Cross-Compiler], page 126.
10. Move the first-stage object files and executables into a subdirectory with this command:

    ```
    make stage1
    ```

    The files are moved into a subdirectory named '`stage1`'. Once installation is complete, you may wish to delete these files with `rm -r stage1`.
11. If you have chosen a configuration for GNU CC which requires other GNU tools (such as GAS or the GNU linker) instead of the standard system tools, install the required tools in the '`stage1`' subdirectory under the names '`as`', '`ld`' or whatever is appropriate. This will enable the stage 1 compiler to find the proper tools in the following stage.

    Alternatively, you can do subsequent compilation using a value of the `PATH` environment variable such that the necessary GNU tools come before the standard system tools.
12. Recompile the compiler with itself, with this command:

    ```
    make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O2"
    ```

    This is called making the stage 2 compiler.

    The command shown above builds compilers for all the supported languages. If you don't want them all, you can specify the languages to build by typing the argument '`LANGUAGES="`*list*`"`'. *list* should contain one or more words from the list '`c`', '`c++`', '`objective-c`', and '`proto`'. Separate the words with spaces. '`proto`' stands for the programs `protoize` and `unprotoize`; they are not a separate language, but you use `LANGUAGES` to enable or disable their installation.

    If you are going to build the stage 3 compiler, then you might want to build only the C language in stage 2.

    Once you have built the stage 2 compiler, if you are short of disk space, you can delete the subdirectory '`stage1`'.

    On a 68000 or 68020 system lacking floating point hardware, unless you have selected a '`tm.h`' file that expects by default that there is no such hardware, do this instead:

    ```
    make CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O2 -msoft-float"
    ```
13. If you wish to test the compiler by compiling it with itself one more time, install any other necessary GNU tools (such as GAS or the GNU linker) in the '`stage2`' subdirectory as you did in the '`stage1`' subdirectory, then do this:

    ```
    make stage2
    make CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O2"
    ```

    This is called making the stage 3 compiler. Aside from the '`-B`' option, the compiler options should be the same as when you made the stage 2 compiler. But the `LANGUAGES` option need not be the same. The command shown above builds compilers for all the supported languages; if you don't want them all, you can specify the languages to build by typing the argument '`LANGUAGES="`*list*`"`', as described above.

If you do not have to install any additional GNU tools, you may use the command

```
make bootstrap LANGUAGES=language-list BOOT_CFLAGS=option-list
```

instead of making '`stage1`', '`stage2`', and performing the two compiler builds.

14. Compare the latest object files with the stage 2 object files—they ought to be identical, aside from time stamps (if any).

    On some systems, meaningful comparison of object files is impossible; they always appear "different." This is currently true on Solaris and some systems that use ELF object file format. On some versions of Irix on SGI machines and DEC Unix (OSF/1) on Alpha systems, you will not be able to compare the files without specifying '`-save-temps`'; see the description of individual systems above to see if you get comparison failures. You may have similar problems on other systems.

    Use this command to compare the files:

    ```
    make compare
    ```

    This will mention any object files that differ between stage 2 and stage 3. Any difference, no matter how innocuous, indicates that the stage 2 compiler has compiled GNU CC incorrectly, and is therefore a potentially serious bug which you should investigate and report (see Chapter 8 [Bugs], page 217).

    If your system does not put time stamps in the object files, then this is a faster way to compare them (using the Bourne shell):

    ```
    for file in *.o; do
    cmp $file stage2/$file
    done
    ```

    If you have built the compiler with the '`-mno-mips-tfile`' option on MIPS machines, you will not be able to compare the files.

15. Install the compiler driver, the compiler's passes and run-time support with '`make install`'. Use the same value for `CC`, `CFLAGS` and `LANGUAGES` that you used when compiling the files that are being installed. One reason this is necessary is that some versions of Make have bugs and recompile files gratuitously when you do this step. If you use the same variable values, those files will be recompiled properly.

    For example, if you have built the stage 2 compiler, you can use the following command:

    ```
    make install CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O" LANGUAGES="list"
    ```

    This copies the files '`cc1`', '`cpp`' and '`libgcc.a`' to files '`cc1`', '`cpp`' and '`libgcc.a`' in the directory '`/usr/local/lib/gcc-lib/`*target*`/`*version*', which is where the compiler driver program looks for them. Here *target* is the canonicalized form of target machine type specified when you ran '`configure`', and *version* is the version number of GNU CC. This naming scheme permits various versions and/or cross-compilers to coexist. It also copies the executables for compilers for other languages (e.g., '`cc1plus`' for C++) to the same directory.

    This also copies the driver program '`xgcc`' into '`/usr/local/bin/gcc`', so that it appears in typical execution search paths. It also copies '`gcc.1`' into '`/usr/local/man/man1`' and info pages into '`/usr/local/info`'.

On some systems, this command causes recompilation of some files. This is usually due to bugs in `make`. You should either ignore this problem, or use GNU Make.

Warning: there is a bug in `alloca` in the Sun library. To avoid this bug, be sure to install the executables of GNU CC that were compiled by GNU CC. (That is, the executables from stage 2 or 3, not stage 1.) They use `alloca` as a built-in function and never the one in the library.

(It is usually better to install GNU CC executables from stage 2 or 3, since they usually run faster than the ones compiled with some other compiler.)

16. If you're going to use C++, you need to install the C++ runtime library. This includes all I/O functionality, special class libraries, etc.

    The standard C++ runtime library for GNU CC is called '`libstdc++`'. An obsolescent library '`libg++`' may also be available, but it's necessary only for older software that hasn't been converted yet; if you don't know whether you need '`libg++`' then you probably don't need it.

    Here's one way to build and install '`libstdc++`' for GNU CC:

    - Build and install GNU CC, so that invoking '`gcc`' obtains the GNU CC that was just built.
    - Obtain a copy of a compatible '`libstdc++`' distribution. For example, the '`libstdc++-2.8.0.tar.gz`' distribution should be compatible with GCC 2.8.0. GCC distributors normally distribute '`libstdc++`' as well.
    - Set the '`CXX`' environment variable to '`gcc`' while running the '`libstdc++`' distribution's '`configure`' command. Use the same '`configure`' options that you used when you invoked GCC's '`configure`' command.
    - Invoke '`make`' to build the C++ runtime.
    - Invoke '`make install`' to install the C++ runtime.

    To summarize, after building and installing GNU CC, invoke the following shell commands in the topmost directory of the C++ library distribution. For *configure-options*, use the same options that you used to configure GNU CC.

    ```
    $ CXX=gcc ./configure configure-options
    $ make
    $ make install
    ```

17. GNU CC includes a runtime library for Objective-C because it is an integral part of the language. You can find the files associated with the library in the subdirectory '`objc`'. The GNU Objective-C Runtime Library requires header files for the target's C library in order to be compiled,and also requires the header files for the target's thread library if you want thread support. See Section 3.4.5 [Cross-Compilers and Header Files], page 129, for discussion about header files issues for cross-compilation.

    When you run '`configure`', it picks the appropriate Objective-C thread implementation file for the target platform. In some situations, you may wish to choose a different back-end as some platforms support multiple thread implementations or you may wish to disable thread support completely. You do this by specifying a value for the *OBJC_THREAD_FILE* makefile variable on the command line when you run make, for example:

```
make CC="stage2/xgcc -Bstage2/" CFLAGS="-g -O2" OBJC_THREAD_FILE=thr-single
```

Below is a list of the currently available back-ends.

- thr-single Disable thread support, should work for all platforms.
- thr-decosf1 DEC OSF/1 thread support.
- thr-irix SGI IRIX thread support.
- thr-mach Generic MACH thread support, known to work on NEXTSTEP.
- thr-os2 IBM OS/2 thread support.
- thr-posix Generix POSIX thread support.
- thr-pthreads PCThreads on Linux-based GNU systems.
- thr-solaris SUN Solaris thread support.
- thr-win32 Microsoft Win32 API thread support.

3.1 Files Created by `configure`

Here we spell out what files will be set up by `configure`. Normally you need not be concerned with these files.

- A file named '`config.h`' is created that contains a '`#include`' of the top-level config file for the machine you will run the compiler on (see section "The Configuration File" in *Using and Porting GCC*). This file is responsible for defining information about the host machine. It includes '`tm.h`'.

  The top-level config file is located in the subdirectory '`config`'. Its name is always '`xm-`*something*`.h`'; usually '`xm-`*machine*`.h`', but there are some exceptions.

  If your system does not support symbolic links, you might want to set up '`config.h`' to contain a '`#include`' command which refers to the appropriate file.
- A file named '`tconfig.h`' is created which includes the top-level config file for your target machine. This is used for compiling certain programs to run on that machine.
- A file named '`tm.h`' is created which includes the machine-description macro file for your target machine. It should be in the subdirectory '`config`' and its name is often '*machine*`.h`'.
- The command file '`configure`' also constructs the file '`Makefile`' by adding some text to the template file '`Makefile.in`'. The additional text comes from files in the '`config`' directory, named '`t-`*target*' and '`x-`*host*'. If these files do not exist, it means nothing needs to be added for a given target or host.

3.2 Configurations Supported by GNU CC

Here are the possible CPU types:

> 1750a, a29k, alpha, arm, c*n*, clipper, dsp16xx, elxsi, h8300, hppa1.0, hppa1.1, i370, i386, i486, i586, i860, i960, m32r, m68000, m68k, m88k, mips, mipsel, mips64, mips64el, ns32k, powerpc, powerpcle, pyramid, romp, rs6000, sh, sparc, sparclite, sparc64, vax, we32k.

Here are the recognized company names. As you can see, customary abbreviations are used rather than the longer official names.

> acorn, alliant, altos, apollo, apple, att, bull, cbm, convergent, convex, crds, dec, dg, dolphin, elxsi, encore, harris, hitachi, hp, ibm, intergraph, isi, mips, motorola, ncr, next, ns, omron, plexus, sequent, sgi, sony, sun, tti, unicom, wrs.

The company name is meaningful only to disambiguate when the rest of the information supplied is insufficient. You can omit it, writing just '*cpu*-*system*', if it is not needed. For example, '`vax-ultrix4.2`' is equivalent to '`vax-dec-ultrix4.2`'.

Here is a list of system types:

> 386bsd, aix, acis, amigaos, aos, aout, aux, bosx, bsd, clix, coff, ctix, cxux, dgux, dynix, ebmon, ecoff, elf, esix, freebsd, hms, genix, gnu, linux-gnu, hiux, hpux, iris, irix, isc, luna, lynxos, mach, minix, msdos, mvs, netbsd, newsos, nindy, ns, osf, osfrose, ptx, riscix, riscos, rtu, sco, sim, solaris, sunos, sym, sysv, udi, ultrix, unicos, uniplus, unos, vms, vsta, vxworks, winnt, xenix.

You can omit the system type; then '`configure`' guesses the operating system from the CPU and company.

You can add a version number to the system type; this may or may not make a difference. For example, you can write '`bsd4.3`' or '`bsd4.4`' to distinguish versions of BSD. In practice, the version number is most needed for '`sysv3`' and '`sysv4`', which are often treated differently.

If you specify an impossible combination such as '`i860-dg-vms`', then you may get an error message from '`configure`', or it may ignore part of the information and do the best it can with the rest. '`configure`' always prints the canonical name for the alternative that it used. GNU CC does not support all possible alternatives.

Often a particular model of machine has a name. Many machine names are recognized as aliases for CPU/company combinations. Thus, the machine name '`sun3`', mentioned above, is an alias for '`m68k-sun`'. Sometimes we accept a company name as a machine name, when the name is popularly used for a particular machine. Here is a table of the known machine names:

> 3300, 3b1, 3b*n*, 7300, altos3068, altos, apollo68, att-7300, balance, convex-c*n*, crds, decstation-3100, decstation, delta, encore, fx2800, gmicro, hp7*nn*, hp8*nn*, hp9k2*nn*, hp9k3*nn*, hp9k7*nn*, hp9k8*nn*, iris4d, iris, isi68, m3230, magnum, merlin, miniframe, mmax, news-3600, news800, news, next, pbd, pc532, pmax, powerpc, powerpcle, ps2, risc-news, rtpc, sun2, sun386i, sun386, sun3, sun4, symmetry, tower-32, tower.

Remember that a machine name specifies both the cpu type and the company name. If you want to install your own homemade configuration files, you can use '`local`' as the company name to access them. If you use configuration '*cpu*`-local`', the configuration name without the cpu prefix is used to form the configuration file names.

Thus, if you specify '`m68k-local`', configuration uses files '`m68k.md`', '`local.h`', '`m68k.c`', '`xm-local.h`', '`t-local`', and '`x-local`', all in the directory '`config/m68k`'.

Here is a list of configurations that have special treatment or special things you must know:

'`1750a-*-*`'

> MIL-STD-1750A processors.
>
> The MIL-STD-1750A cross configuration produces output for `as1750`, an assembler/linker available under the GNU Public License for the 1750A. `as1750`

can be obtained at *ftp://ftp.fta-berlin.de/pub/crossgcc/1750gals/*. A similarly licensed simulator for the 1750A is available from same address.

You should ignore a fatal error during the building of libgcc (libgcc is not yet implemented for the 1750A.)

The `as1750` assembler requires the file '`ms1750.inc`', which is found in the directory '`config/1750a`'.

GNU CC produced the same sections as the Fairchild F9450 C Compiler, namely:

`Normal`	The program code section.
`Static`	The read/write (RAM) data section.
`Konst`	The read-only (ROM) constants section.
`Init`	Initialization section (code to copy KREL to SREL).

The smallest addressable unit is 16 bits (BITS_PER_UNIT is 16). This means that type 'char' is represented with a 16-bit word per character. The 1750A's "Load/Store Upper/Lower Byte" instructions are not used by GNU CC.

'`alpha-*-osf1`'

Systems using processors that implement the DEC Alpha architecture and are running the DEC Unix (OSF/1) operating system, for example the DEC Alpha AXP systems.CC.)

GNU CC writes a '`.verstamp`' directive to the assembler output file unless it is built as a cross-compiler. It gets the version to use from the system header file '`/usr/include/stamp.h`'. If you install a new version of DEC Unix, you should rebuild GCC to pick up the new version stamp.

Note that since the Alpha is a 64-bit architecture, cross-compilers from 32-bit machines will not generate code as efficient as that generated when the compiler is running on a 64-bit machine because many optimizations that depend on being able to represent a word on the target in an integral value on the host cannot be performed. Building cross-compilers on the Alpha for 32-bit machines has only been tested in a few cases and may not work properly.

`make compare` may fail on old versions of DEC Unix unless you add '`-save-temps`' to `CFLAGS`. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the `stage1` and `stage2` compilations. The option '`-save-temps`' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in '`/tmp`'. Do not add '`-save-temps`' unless the comparisons fail without that option. If you add '`-save-temps`', you will have to manually delete the '`.i`' and '`.s`' files after each series of compilations.

GNU CC now supports both the native (ECOFF) debugging format used by DBX and GDB and an encapsulated STABS format for use only with GDB. See the discussion of the '`--with-stabs`' option of '`configure`' above for more information on these formats and how to select them.

There is a bug in DEC's assembler that produces incorrect line numbers for ECOFF format when the '`.align`' directive is used. To work around this problem, GNU CC will not emit such alignment directives while writing ECOFF format debugging information even if optimization is being performed. Unfortunately, this has the very undesirable side-effect that code addresses when '`-O`' is specified are different depending on whether or not '`-g`' is also specified.

To avoid this behavior, specify '`-gstabs+`' and use GDB instead of DBX. DEC is now aware of this problem with the assembler and hopes to provide a fix shortly.

'`arc-*-elf`'
: Argonaut ARC processor. This configuration is intended for embedded systems.

'`arm-*-aout`'
: Advanced RISC Machines ARM-family processors. These are often used in embedded applications. There are no standard Unix configurations. This configuration corresponds to the basic instruction sequences and will produce '`a.out`' format object modules.

  You may need to make a variant of the file '`arm.h`' for your particular configuration.

'`arm-*-linuxaout`'
: Any of the ARM family processors running the Linux-based GNU system with the '`a.out`' binary format (ELF is not yet supported). You must use version 2.8.1.0.7 or later of the GNU/Linux binutils, which you can download from '`sunsite.unc.edu:/pub/Linux/GCC`' and other mirror sites for Linux-based GNU systems.

'`arm-*-riscix`'
: The ARM2 or ARM3 processor running RISC iX, Acorn's port of BSD Unix. If you are running a version of RISC iX prior to 1.2 then you must specify the version number during configuration. Note that the assembler shipped with RISC iX does not support stabs debugging information; a new version of the assembler, with stabs support included, is now available from Acorn and via ftp '`ftp.acorn.com:/pub/riscix/as+xterm.tar.Z`'. To enable stabs debugging, pass '`--with-gnu-as`' to configure.

  You will need to install GNU '`sed`' before you can run configure.

'`a29k`'
: AMD Am29k-family processors. These are normally used in embedded applications. There are no standard Unix configurations. This configuration corresponds to AMD's standard calling sequence and binary interface and is compatible with other 29k tools.

  You may need to make a variant of the file '`a29k.h`' for your particular configuration.

'`a29k-*-bsd`'
: AMD Am29050 used in a system running a variant of BSD Unix.

'`decstation-*`'
: MIPS-based DECstations can support three different personalities: Ultrix, DEC OSF/1, and OSF/rose. (Alpha-based DECstation products have a configuration name beginning with '`alpha-dec`'.) To configure GCC for these platforms use the following configurations:

'decstation-ultrix'
: Ultrix configuration.

'decstation-osf1'
: Dec's version of OSF/1.

'decstation-osfrose'
: Open Software Foundation reference port of OSF/1 which uses the OSF/rose object file format instead of ECOFF. Normally, you would not select this configuration.

The MIPS C compiler needs to be told to increase its table size for switch statements with the '-Wf,-XNg1500' option in order to compile 'cp/parse.c'. If you use the '-O2' optimization option, you also need to use '-Olimit 3000'. Both of these options are automatically generated in the 'Makefile' that the shell script 'configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

'elxsi-elxsi-bsd'
: The Elxsi's C compiler has known limitations that prevent it from compiling GNU C. Please contact mrs@cygnus.com for more details.

'dsp16xx' A port to the AT&T DSP1610 family of processors.

'h8300-*-*'
: Hitachi H8/300 series of processors.

The calling convention and structure layout has changed in release 2.6. All code must be recompiled. The calling convention now passes the first three arguments in function calls in registers. Structures are no longer a multiple of 2 bytes.

'hppa*-*-*'
: There are several variants of the HP-PA processor which run a variety of operating systems. GNU CC must be configured to use the correct processor type and operating system, or GNU CC will not function correctly. The easiest way to handle this problem is to *not* specify a target when configuring GNU CC, the 'configure' script will try to automatically determine the right processor type and operating system.

'-g' does not work on HP-UX, since that system uses a peculiar debugging format which GNU CC does not know about. However, '-g' will work if you also use GAS and GDB in conjunction with GCC. We highly recommend using GAS for all HP-PA configurations.

You should be using GAS-2.6 (or later) along with GDB-4.16 (or later). These can be retrieved from all the traditional GNU ftp archive sites.

On some versions of HP-UX, you will need to install GNU 'sed'.

You will need to be install GAS into a directory before /bin, /usr/bin, and /usr/ccs/bin in your search path. You should install GAS before you build GNU CC.

To enable debugging, you must configure GNU CC with the '--with-gnu-as' option before building.

'i370-*-*'
: This port is very preliminary and has many known bugs. We hope to have a higher-quality port for this machine soon.

'i386-*-linux-gnuoldld'
: Use this configuration to generate 'a.out' binaries on Linux-based GNU systems if you do not have gas/binutils version 2.5.2 or later installed. This is an obsolete configuration.

'i386-*-linux-gnuaout'
: Use this configuration to generate 'a.out' binaries on Linux-based GNU systems. This configuration is being superseded. You must use gas/binutils version 2.5.2 or later.

'i386-*-linux-gnu'
: Use this configuration to generate ELF binaries on Linux-based GNU systems. You must use gas/binutils version 2.5.2 or later.

'i386-*-sco'
: Compilation with RCC is recommended. Also, it may be a good idea to link with GNU malloc instead of the malloc that comes with the system.

'i386-*-sco3.2v4'
: Use this configuration for SCO release 3.2 version 4.

'i386-*-sco3.2v5*'
: Use this for the SCO OpenServer Release family including 5.0.0, 5.0.2, 5.0.4, 5.0.5, Internet FastStart 1.0, and Internet FastStart 1.1.

  GNU CC can generate COFF binaries if you specify '-mcoff' or ELF binaries, the default. A full 'make bootstrap' is recommended so that an ELF compiler that builds ELF is generated.

  You must have TLS597 from ftp://ftp.sco.com/TLS installed for ELF C++ binaries to work correctly on releases before 5.0.4.

  The native SCO assembler that is provided with the OS at no charge is normally required. If, however, you must be able to use the GNU assembler (perhaps you have complex asms) you must configure this package '--with-gnu-as'. To do this, install (cp or symlink) gcc/as to your copy of the GNU assembler. You must use a recent version of GNU binutils; version 2.9.1 seems to work well. If you select this option, you will be unable to build COFF images. Trying to do so will result in non-obvious failures. In general, the "–with-gnu-as" option isn't as well tested as the native assembler.

  NOTE: If you are building C++, you must follow the instructions about invoking 'make bootstrap' because the native OpenServer compiler may build a 'cc1plus' that will not correctly parse many valid C++ programs. You must do a 'make bootstrap' if you are building with the native compiler.

'i386-*-isc'
: It may be a good idea to link with GNU malloc instead of the malloc that comes with the system.

  In ISC version 4.1, 'sed' core dumps when building 'deduced.h'. Use the version of 'sed' from version 4.0.

'`i386-*-esix`'
: It may be good idea to link with GNU malloc instead of the malloc that comes with the system.

'`i386-ibm-aix`'
: You need to use GAS version 2.1 or later, and LD from GNU binutils version 2.2 or later.

'`i386-sequent-bsd`'
: Go to the Berkeley universe before compiling.

'`i386-sequent-ptx1*`'
'`i386-sequent-ptx2*`'
: You must install GNU '`sed`' before running '`configure`'.

'`i386-sun-sunos4`'
: You may find that you need another version of GNU CC to begin bootstrapping with, since the current version when built with the system's own compiler seems to get an infinite loop compiling part of '`libgcc2.c`'. GNU CC version 2 compiled with GNU CC (any version) seems not to have this problem.

  See Section 3.5 [Sun Install], page 131, for information on installing GNU CC on Sun systems.

'`i[345]86-*-winnt3.5`'
: This version requires a GAS that has not yet been released. Until it is, you can get a prebuilt binary version via anonymous ftp from '`cs.washington.edu:pub/gnat`' or '`cs.nyu.edu:pub/gnat`'. You must also use the Microsoft header files from the Windows NT 3.5 SDK. Find these on the CDROM in the '`/mstools/h`' directory dated 9/4/94. You must use a fixed version of Microsoft linker made especially for NT 3.5, which is also is available on the NT 3.5 SDK CDROM. If you do not have this linker, can you also use the linker from Visual C/C++ 1.0 or 2.0.

  Installing GNU CC for NT builds a wrapper linker, called '`ld.exe`', which mimics the behaviour of Unix '`ld`' in the specification of libraries ('`-L`' and '`-l`'). '`ld.exe`' looks for both Unix and Microsoft named libraries. For example, if you specify '`-lfoo`', '`ld.exe`' will look first for '`libfoo.a`' and then for '`foo.lib`'.

  You may install GNU CC for Windows NT in one of two ways, depending on whether or not you have a Unix-like shell and various Unix-like utilities.

  1. If you do not have a Unix-like shell and few Unix-like utilities, you will use a DOS style batch script called '`configure.bat`'. Invoke it as `configure winnt` from an MSDOS console window or from the program manager dialog box. '`configure.bat`' assumes you have already installed and have in your path a Unix-like '`sed`' program which is used to create a working '`Makefile`' from '`Makefile.in`'.

     '`Makefile`' uses the Microsoft Nmake program maintenance utility and the Visual C/C++ V8.00 compiler to build GNU CC. You need only have the utilities '`sed`' and '`touch`' to use this installation method, which only automatically builds the compiler itself. You must then examine what '`fixinc.winnt`' does, edit the header files by hand and build '`libgcc.a`' manually.

2. The second type of installation assumes you are running a Unix-like shell, have a complete suite of Unix-like utilities in your path, and have a previous version of GNU CC already installed, either through building it via the above installation method or acquiring a pre-built binary. In this case, use the '`configure`' script in the normal fashion.

'`i860-intel-osf1`'
: This is the Paragon. If you have version 1.0 of the operating system, see Section 7.2 [Installation Problems], page 193, for special things you need to do to compensate for peculiarities in the system.

'`*-lynx-lynxos`'
: LynxOS 2.2 and earlier comes with GNU CC 1.x already installed as '`/bin/gcc`'. You should compile with this instead of '`/bin/cc`'. You can tell GNU CC to use the GNU assembler and linker, by specifying '`--with-gnu-as --with-gnu-ld`' when configuring. These will produce COFF format object files and executables; otherwise GNU CC will use the installed tools, which produce '`a.out`' format executables.

'`m32r-*-elf`'
: Mitsubishi M32R processor. This configuration is intended for embedded systems.

'`m68000-hp-bsd`'
: HP 9000 series 200 running BSD. Note that the C compiler that comes with this system cannot compile GNU CC; contact `law@cygnus.com` to get binaries of GNU CC for bootstrapping.

'`m68k-altos`'
: Altos 3068. You must use the GNU assembler, linker and debugger. Also, you must fix a kernel bug. Details in the file '`README.ALTOS`'.

'`m68k-apple-aux`'
: Apple Macintosh running A/UX. You may configure GCC to use either the system assembler and linker or the GNU assembler and linker. You should use the GNU configuration if you can, especially if you also want to use GNU C++. You enabled that configuration with + the '`--with-gnu-as`' and '`--with-gnu-ld`' options to `configure`.

  Note the C compiler that comes with this system cannot compile GNU CC. You can find binaries of GNU CC for bootstrapping on `jagubox.gsfc.nasa.gov`. You will also a patched version of '`/bin/ld`' there that raises some of the arbitrary limits found in the original.

'`m68k-att-sysv`'
: AT&T 3b1, a.k.a. 7300 PC. Special procedures are needed to compile GNU CC with this machine's standard C compiler, due to bugs in that compiler. You can bootstrap it more easily with previous versions of GNU CC if you have them.

Installing GNU CC on the 3b1 is difficult if you do not already have GNU CC running, due to bugs in the installed C compiler. However, the following procedure might work. We are unable to test it.

1. Comment out the '`#include "config.h"`' line near the start of '`cccp.c`' and do '`make cpp`'. This makes a preliminary version of GNU cpp.
2. Save the old '`/lib/cpp`' and copy the preliminary GNU cpp to that file name.
3. Undo your change in '`cccp.c`', or reinstall the original version, and do '`make cpp`' again.
4. Copy this final version of GNU cpp into '`/lib/cpp`'.
5. Replace every occurrence of `obstack_free` in the file '`tree.c`' with `_obstack_free`.
6. Run `make` to get the first-stage GNU CC.
7. Reinstall the original version of '`/lib/cpp`'.
8. Now you can compile GNU CC with itself and install it in the normal fashion.

'`m68k-bull-sysv`'

Bull DPX/2 series 200 and 300 with BOS-2.00.45 up to BOS-2.01. GNU CC works either with native assembler or GNU assembler. You can use GNU assembler with native coff generation by providing '`--with-gnu-as`' to the configure script or use GNU assembler with dbx-in-coff encapsulation by providing '`--with-gnu-as --stabs`'. For any problem with native assembler or for availability of the DPX/2 port of GAS, contact `F.Pierresteguy@frcl.bull.fr`.

'`m68k-crds-unox`'

Use '`configure unos`' for building on Unos.

The Unos assembler is named `casm` instead of `as`. For some strange reason linking '`/bin/as`' to '`/bin/casm`' changes the behavior, and does not work. So, when installing GNU CC, you should install the following script as '`as`' in the subdirectory where the passes of GCC are installed:

```
#!/bin/sh
casm $*
```

The default Unos library is named '`libunos.a`' instead of '`libc.a`'. To allow GNU CC to function, either change all references to '`-lc`' in '`gcc.c`' to '`-lunos`' or link '`/lib/libc.a`' to '`/lib/libunos.a`'.

When compiling GNU CC with the standard compiler, to overcome bugs in the support of `alloca`, do not use '`-O`' when making stage 2. Then use the stage 2 compiler with '`-O`' to make the stage 3 compiler. This compiler will have the same characteristics as the usual stage 2 compiler on other systems. Use it to make a stage 4 compiler and compare that with stage 3 to verify proper compilation.

(Perhaps simply defining `ALLOCA` in '`x-crds`' as described in the comments there will make the above paragraph superfluous. Please inform us of whether this works.)

Unos uses memory segmentation instead of demand paging, so you will need a lot of memory. 5 Mb is barely enough if no other tasks are running. If linking '`cc1`' fails, try putting the object files into a library and linking from that library.

'`m68k-hp-hpux`'

HP 9000 series 300 or 400 running HP-UX. HP-UX version 8.0 has a bug in the assembler that prevents compilation of GNU CC. To fix it, get patch PHCO_4484 from HP.

In addition, if you wish to use gas '`--with-gnu-as`' you must use gas version 2.1 or later, and you must use the GNU linker version 2.1 or later. Earlier versions of gas relied upon a program which converted the gas output into the native HP-UX format, but that program has not been kept up to date. gdb does not understand that native HP-UX format, so you must use gas if you wish to use gdb.

'`m68k-sun`'

Sun 3. We do not provide a configuration file to use the Sun FPA by default, because programs that establish signal handlers for floating point traps inherently cannot work with the FPA.

See Section 3.5 [Sun Install], page 131, for information on installing GNU CC on Sun systems.

'`m88k-*-svr3`'

Motorola m88k running the AT&T/Unisoft/Motorola V.3 reference port. These systems tend to use the Green Hills C, revision 1.8.5, as the standard C compiler. There are apparently bugs in this compiler that result in object files differences between stage 2 and stage 3. If this happens, make the stage 4 compiler and compare it to the stage 3 compiler. If the stage 3 and stage 4 object files are identical, this suggests you encountered a problem with the standard C compiler; the stage 3 and 4 compilers may be usable.

It is best, however, to use an older version of GNU CC for bootstrapping if you have one.

'`m88k-*-dgux`'

Motorola m88k running DG/UX. To build 88open BCS native or cross compilers on DG/UX, specify the configuration name as '`m88k-*-dguxbcs`' and build in the 88open BCS software development environment. To build ELF native or cross compilers on DG/UX, specify '`m88k-*-dgux`' and build in the DG/UX ELF development environment. You set the software development environment by issuing '`sde-target`' command and specifying either '`m88kbcs`' or '`m88kdguxelf`' as the operand.

If you do not specify a configuration name, '`configure`' guesses the configuration based on the current software development environment.

'`m88k-tektronix-sysv3`'

Tektronix XD88 running UTekV 3.2e. Do not turn on optimization while building stage1 if you bootstrap with the buggy Green Hills compiler. Also, The bundled LAI System V NFS is buggy so if you build in an NFS mounted directory, start from a fresh reboot, or avoid NFS all together. Otherwise you may have trouble getting clean comparisons between stages.

'`mips-mips-bsd`'

MIPS machines running the MIPS operating system in BSD mode. It's possible that some old versions of the system lack the functions `memcpy`, `memcmp`, and `memset`. If your system lacks these, you must remove or undo the definition of `TARGET_MEM_FUNCTIONS` in '`mips-bsd.h`'.

The MIPS C compiler needs to be told to increase its table size for switch statements with the '`-Wf,-XNg1500`' option in order to compile '`cp/parse.c`'. If you use the '`-O2`' optimization option, you also need to use '`-Olimit 3000`'. Both of these options are automatically generated in the '`Makefile`' that the shell script '`configure`' builds. If you override the `CC` make variable and use the MIPS compilers, you may need to add '`-Wf,-XNg1500 -Olimit 3000`'.

'`mips-mips-riscos*`'

The MIPS C compiler needs to be told to increase its table size for switch statements with the '`-Wf,-XNg1500`' option in order to compile '`cp/parse.c`'. If you use the '`-O2`' optimization option, you also need to use '`-Olimit 3000`'. Both of these options are automatically generated in the '`Makefile`' that the shell script '`configure`' builds. If you override the `CC` make variable and use the MIPS compilers, you may need to add '`-Wf,-XNg1500 -Olimit 3000`'.

MIPS computers running RISC-OS can support four different personalities: default, BSD 4.3, System V.3, and System V.4 (older versions of RISC-OS don't support V.4). To configure GCC for these platforms use the following configurations:

'`mips-mips-riscosrev`'

Default configuration for RISC-OS, revision `rev`.

'`mips-mips-riscosrevbsd`'

BSD 4.3 configuration for RISC-OS, revision `rev`.

'`mips-mips-riscosrevsysv4`'

System V.4 configuration for RISC-OS, revision `rev`.

'`mips-mips-riscosrevsysv`'

System V.3 configuration for RISC-OS, revision `rev`.

The revision `rev` mentioned above is the revision of RISC-OS to use. You must reconfigure GCC when going from a RISC-OS revision 4 to RISC-OS revision 5. This has the effect of avoiding a linker bug (see Section 7.2 [Installation Problems], page 193, for more details).

'`mips-sgi-*`'

In order to compile GCC on an SGI running IRIX 4, the "c.hdr.lib" option must be installed from the CD-ROM supplied from Silicon Graphics. This is found on the 2nd CD in release 4.0.1.

In order to compile GCC on an SGI running IRIX 5, the "compiler_dev.hdr" subsystem must be installed from the IDO CD-ROM supplied by Silicon Graphics.

make compare may fail on version 5 of IRIX unless you add '-save-temps' to CFLAGS. On these systems, the name of the assembler input file is stored in the object file, and that makes comparison fail if it differs between the stage1 and stage2 compilations. The option '-save-temps' forces a fixed name to be used for the assembler input file, instead of a randomly chosen name in '/tmp'. Do not add '-save-temps' unless the comparisons fail without that option. If you do you '-save-temps', you will have to manually delete the '.i' and '.s' files after each series of compilations.

The MIPS C compiler needs to be told to increase its table size for switch statements with the '-Wf,-XNg1500' option in order to compile 'cp/parse.c'. If you use the '-O2' optimization option, you also need to use '-Olimit 3000'. Both of these options are automatically generated in the 'Makefile' that the shell script 'configure' builds. If you override the CC make variable and use the MIPS compilers, you may need to add '-Wf,-XNg1500 -Olimit 3000'.

On Irix version 4.0.5F, and perhaps on some other versions as well, there is an assembler bug that reorders instructions incorrectly. To work around it, specify the target configuration 'mips-sgi-irix4loser'. This configuration inhibits assembler optimization.

In a compiler configured with target 'mips-sgi-irix4', you can turn off assembler optimization by using the '-noasmopt' option. This compiler option passes the option '-O0' to the assembler, to inhibit reordering.

The '-noasmopt' option can be useful for testing whether a problem is due to erroneous assembler reordering. Even if a problem does not go away with '-noasmopt', it may still be due to assembler reordering—perhaps GNU CC itself was miscompiled as a result.

To enable debugging under Irix 5, you must use GNU as 2.5 or later, and use the '--with-gnu-as' configure option when configuring gcc. GNU as is distributed as part of the binutils package.

'mips-sony-sysv'
: Sony MIPS NEWS. This works in NEWSOS 5.0.1, but not in 5.0.2 (which uses ELF instead of COFF). Support for 5.0.2 will probably be provided soon by volunteers. In particular, the linker does not like the code generated by GCC when shared libraries are linked in.

'ns32k-encore'
: Encore ns32000 system. Encore systems are supported only under BSD.

'ns32k-*-genix'
: National Semiconductor ns32000 system. Genix has bugs in alloca and malloc; you must get the compiled versions of these from GNU Emacs.

'ns32k-sequent'
: Go to the Berkeley universe before compiling.

'ns32k-utek'
: UTEK ns32000 system ("merlin"). The C compiler that comes with this system cannot compile GNU CC; contact 'tektronix!reed!mason' to get binaries of GNU CC for bootstrapping.

'`romp-*-aos`'
'`romp-*-mach`'

The only operating systems supported for the IBM RT PC are AOS and MACH. GNU CC does not support AIX running on the RT. We recommend you compile GNU CC with an earlier version of itself; if you compile GNU CC with `hc`, the Metaware compiler, it will work, but you will get mismatches between the stage 2 and stage 3 compilers in various files. These errors are minor differences in some floating-point constants and can be safely ignored; the stage 3 compiler is correct.

'`rs6000-*-aix`'
'`powerpc-*-aix`'

Various early versions of each release of the IBM XLC compiler will not bootstrap GNU CC. Symptoms include differences between the stage2 and stage3 object files, and errors when compiling '`libgcc.a`' or '`enquire`'. Known problematic releases include: xlc-1.2.1.8, xlc-1.3.0.0 (distributed with AIX 3.2.5), and xlc-1.3.0.19. Both xlc-1.2.1.28 and xlc-1.3.0.24 (PTF 432238) are known to produce working versions of GNU CC, but most other recent releases correctly bootstrap GNU CC.

Release 4.3.0 of AIX and ones prior to AIX 3.2.4 include a version of the IBM assembler which does not accept debugging directives: assembler updates are available as PTFs. Also, if you are using AIX 3.2.5 or greater and the GNU assembler, you must have a version modified after October 16th, 1995 in order for the GNU C compiler to build. See the file '`README.RS6000`' for more details on any of these problems.

GNU CC does not yet support the 64-bit PowerPC instructions.

Objective C does not work on this architecture because it makes assumptions that are incompatible with the calling conventions.

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 GNU CC 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".

Due to changes in the way that GNU CC invokes the binder (linker) for AIX 4.1, you may now receive warnings of duplicate symbols from the link step that were not reported before. The assembly files generated by GNU CC for AIX have always included multiple symbol definitions for certain global variable and function declarations in the original program. The warnings should not prevent the linker from producing a correct library or runnable executable.

By default, AIX 4.1 produces code that can be used on either Power or PowerPC processors.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpc-*-elf`'
'`powerpc-*-sysv4`'

PowerPC system in big endian mode, running System V.4.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpc-*-linux-gnu`'

PowerPC system in big endian mode, running the Linux-based GNU system.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpc-*-eabiaix`'

Embedded PowerPC system in big endian mode with -mcall-aix selected as the default.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpc-*-eabisim`'

Embedded PowerPC system in big endian mode for use in running under the PSIM simulator.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpc-*-eabi`'

Embedded PowerPC system in big endian mode.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpcle-*-elf`'
'`powerpcle-*-sysv4`'

PowerPC system in little endian mode, running System V.4.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

'`powerpcle-*-solaris2*`'

PowerPC system in little endian mode, running Solaris 2.5.1 or higher.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*. Beta versions of the Sun 4.0 compiler do not seem to be able to build GNU CC correctly. There are also problems with the host assembler and linker that are fixed by using the GNU versions of these tools.

'`powerpcle-*-eabisim`'

Embedded PowerPC system in little endian mode for use in running under the PSIM simulator.

'`powerpcle-*-eabi`'

Embedded PowerPC system in little endian mode.

You can specify a default version for the '`-mcpu=`'*cpu_type* switch by using the configure option '`--with-cpu-`'*cpu_type*.

‘`powerpcle-*-winnt`’
‘`powerpcle-*-pe`’

PowerPC system in little endian mode running Windows NT.

You can specify a default version for the ‘`-mcpu=`’*cpu_type* switch by using the configure option ‘`--with-cpu-`’*cpu_type*.

‘`vax-dec-ultrix`’

Don’t try compiling with Vax C (`vcc`). It produces incorrect code in some cases (for example, when `alloca` is used).

Meanwhile, compiling ‘`cp/parse.c`’ with pcc does not work because of an internal table size limitation in that compiler. To avoid this problem, compile just the GNU C compiler first, and use it to recompile building all the languages that you want to run.

‘`sparc-sun-*`’

See Section 3.5 [Sun Install], page 131, for information on installing GNU CC on Sun systems.

‘`vax-dec-vms`’

See Section 3.6 [VMS Install], page 131, for details on how to install GNU CC on VMS.

‘`we32k-*-*`’

These computers are also known as the 3b2, 3b5, 3b20 and other similar names. (However, the 3b1 is actually a 68000; see Section 3.2 [Configurations], page 111.)

Don’t use ‘`-g`’ when compiling with the system’s compiler. The system’s linker seems to be unable to handle such a large program with debugging information.

The system’s compiler runs out of capacity when compiling ‘`stmt.c`’ in GNU CC. You can work around this by building ‘`cpp`’ in GNU CC first, then use that instead of the system’s preprocessor with the system’s C compiler to compile ‘`stmt.c`’. Here is how:

```
mv /lib/cpp /lib/cpp.att
cp cpp /lib/cpp.gnu
echo '/lib/cpp.gnu -traditional ${1+"$@"}' > /lib/cpp
chmod +x /lib/cpp
```

The system’s compiler produces bad code for some of the GNU CC optimization files. So you must build the stage 2 compiler without optimization. Then build a stage 3 compiler with optimization. That executable should work. Here are the necessary commands:

```
make LANGUAGES=c CC=stage1/xgcc CFLAGS="-Bstage1/ -g"
make stage2
make CC=stage2/xgcc CFLAGS="-Bstage2/ -g -O"
```

You may need to raise the ULIMIT setting to build a C++ compiler, as the file ‘`cc1plus`’ is larger than one megabyte.

3.3 Compilation in a Separate Directory

If you wish to build the object files and executables in a directory other than the one containing the source files, here is what you must do differently:

1. Make sure you have a version of Make that supports the `VPATH` feature. (GNU Make supports it, as do Make versions on most BSD systems.)
2. If you have ever run '`configure`' in the source directory, you must undo the configuration. Do this by running:

   ```
   make distclean
   ```
3. Go to the directory in which you want to build the compiler before running '`configure`':

   ```
   mkdir gcc-sun3
   cd gcc-sun3
   ```

   On systems that do not support symbolic links, this directory must be on the same file system as the source code directory.
4. Specify where to find '`configure`' when you run it:

   ```
   ../gcc/configure ...
   ```

   This also tells `configure` where to find the compiler sources; `configure` takes the directory from the file name that was used to invoke it. But if you want to be sure, you can specify the source directory with the '`--srcdir`' option, like this:

   ```
   ../gcc/configure --srcdir=../gcc other options
   ```

   The directory you specify with '`--srcdir`' need not be the same as the one that `configure` is found in.

Now, you can run `make` in that directory. You need not repeat the configuration steps shown above, when ordinary source files change. You must, however, run `configure` again when the configuration files change, if your system does not support symbolic links.

3.4 Building and Installing a Cross-Compiler

GNU CC can function as a cross-compiler for many machines, but not all.

- Cross-compilers for the Mips as target using the Mips assembler currently do not work, because the auxiliary programs '`mips-tdump.c`' and '`mips-tfile.c`' can't be compiled on anything but a Mips. It does work to cross compile for a Mips if you use the GNU assembler and linker.
- Cross-compilers between machines with different floating point formats have not all been made to work. GNU CC now has a floating point emulator with which these can work, but each target machine description needs to be updated to take advantage of it.
- Cross-compilation between machines of different word sizes is somewhat problematic and sometimes does not work.

Since GNU CC generates assembler code, you probably need a cross-assembler that GNU CC can run, in order to produce object files. If you want to link on other than the target machine, you need a cross-linker as well. You also need header files and libraries suitable for the target machine that you can install on the host machine.

3.4.1 Steps of Cross-Compilation

To compile and run a program using a cross-compiler involves several steps:

- Run the cross-compiler on the host machine to produce assembler files for the target machine. This requires header files for the target machine.
- Assemble the files produced by the cross-compiler. You can do this either with an assembler on the target machine, or with a cross-assembler on the host machine.
- Link those files to make an executable. You can do this either with a linker on the target machine, or with a cross-linker on the host machine. Whichever machine you use, you need libraries and certain startup files (typically '`crt....o`') for the target machine.

It is most convenient to do all of these steps on the same host machine, since then you can do it all with a single invocation of GNU CC. This requires a suitable cross-assembler and cross-linker. For some targets, the GNU assembler and linker are available.

3.4.2 Configuring a Cross-Compiler

To build GNU CC as a cross-compiler, you start out by running '`configure`'. Use the '`--target=`*target*' to specify the target type. If '`configure`' was unable to correctly identify the system you are running on, also specify the '`--build=`*build*' option. For example, here is how to configure for a cross-compiler that produces code for an HP 68030 system running BSD on a system that '`configure`' can correctly identify:

```
./configure --target=m68k-hp-bsd4.3
```

3.4.3 Tools and Libraries for a Cross-Compiler

If you have a cross-assembler and cross-linker available, you should install them now. Put them in the directory '`/usr/local/`*target*`/bin`'. Here is a table of the tools you should put in this directory:

'`as`'	This should be the cross-assembler.
'`ld`'	This should be the cross-linker.
'`ar`'	This should be the cross-archiver: a program which can manipulate archive files (linker libraries) in the target machine's format.
'`ranlib`'	This should be a program to construct a symbol table in an archive file.

The installation of GNU CC will find these programs in that directory, and copy or link them to the proper place to for the cross-compiler to find them when run later.

The easiest way to provide these files is to build the Binutils package and GAS. Configure them with the same '`--host`' and '`--target`' options that you use for configuring GNU CC, then build and install them. They install their executables automatically into the proper directory. Alas, they do not support all the targets that GNU CC supports.

If you want to install libraries to use with the cross-compiler, such as a standard C library, put them in the directory '`/usr/local/`*target*`/lib`'; installation of GNU CC copies all the files in that subdirectory into the proper place for GNU CC to find them and link with them. Here's an example of copying some libraries from a target machine:

```
ftp target-machine
lcd /usr/local/target/lib
cd /lib
get libc.a
cd /usr/lib
get libg.a
get libm.a
quit
```

The precise set of libraries you'll need, and their locations on the target machine, vary depending on its operating system.

Many targets require "start files" such as '`crt0.o`' and '`crtn.o`' which are linked into each executable; these too should be placed in '`/usr/local/`*target*`/lib`'. There may be several alternatives for '`crt0.o`', for use with profiling or other compilation options. Check your target's definition of `STARTFILE_SPEC` to find out what start files it uses. Here's an example of copying these files from a target machine:

```
ftp target-machine
lcd /usr/local/target/lib
prompt
cd /lib
mget *crt*.o
cd /usr/lib
mget *crt*.o
quit
```

3.4.4 '`libgcc.a`' and Cross-Compilers

Code compiled by GNU CC uses certain runtime support functions implicitly. Some of these functions can be compiled successfully with GNU CC itself, but a few cannot be. These problem functions are in the source file '`libgcc1.c`'; the library made from them is called '`libgcc1.a`'.

When you build a native compiler, these functions are compiled with some other compiler–the one that you use for bootstrapping GNU CC. Presumably it knows how to open code these operations, or else knows how to call the run-time emulation facilities that the machine comes with. But this approach doesn't work for building a cross-compiler. The compiler that you use for building knows about the host system, not the target system.

So, when you build a cross-compiler you have to supply a suitable library '`libgcc1.a`' that does the job it is expected to do.

To compile '`libgcc1.c`' with the cross-compiler itself does not work. The functions in this file are supposed to implement arithmetic operations that GNU CC does not know how to open code for your target machine. If these functions are compiled with GNU CC itself, they will compile into infinite recursion.

On any given target, most of these functions are not needed. If GNU CC can open code an arithmetic operation, it will not call these functions to perform the operation. It is possible that on your target machine, none of these functions is needed. If so, you can supply an empty library as '`libgcc1.a`'.

Many targets need library support only for multiplication and division. If you are linking with a library that contains functions for multiplication and division, you can tell GNU CC to call them directly by defining the macros `MULSI3_LIBCALL`, and the like. These macros need to be defined in the target description macro file. For some targets, they are defined already. This may be sufficient to avoid the need for libgcc1.a; if so, you can supply an empty library.

Some targets do not have floating point instructions; they need other functions in '`libgcc1.a`', which do floating arithmetic. Recent versions of GNU CC have a file which emulates floating point. With a certain amount of work, you should be able to construct a floating point emulator that can be used as '`libgcc1.a`'. Perhaps future versions will contain code to do this automatically and conveniently. That depends on whether someone wants to implement it.

Some embedded targets come with all the necessary '`libgcc1.a`' routines written in C or assembler. These targets build '`libgcc1.a`' automatically and you do not need to do anything special for them. Other embedded targets do not need any '`libgcc1.a`' routines since all the necessary operations are supported by the hardware.

If your target system has another C compiler, you can configure GNU CC as a native compiler on that machine, build just '`libgcc1.a`' with '`make libgcc1.a`' on that machine, and use the resulting file with the cross-compiler. To do this, execute the following on the target machine:

```
cd target-build-dir
./configure --host=sparc --target=sun3
make libgcc1.a
```

And then this on the host machine:

```
ftp target-machine
binary
cd target-build-dir
get libgcc1.a
quit
```

Another way to provide the functions you need in '`libgcc1.a`' is to define the appropriate `perform_...` macros for those functions. If these definitions do not use the C arithmetic operators that they are meant to implement, you should be able to compile them with the cross-compiler you are building. (If these definitions already exist for your target file, then you are all set.)

To build '`libgcc1.a`' using the perform macros, use '`LIBGCC1=libgcc1.a OLDCC=./xgcc`' when building the compiler. Otherwise, you should place your replacement library under the name '`libgcc1.a`' in the directory in which you will build the cross-compiler, before you run `make`.

3.4.5 Cross-Compilers and Header Files

If you are cross-compiling a standalone program or a program for an embedded system, then you may not need any header files except the few that are part of GNU CC (and those of your program). However, if you intend to link your program with a standard C library such as '`libc.a`', then you probably need to compile with the header files that go with the library you use.

The GNU C compiler does not come with these files, because (1) they are system-specific, and (2) they belong in a C library, not in a compiler.

If the GNU C library supports your target machine, then you can get the header files from there (assuming you actually use the GNU library when you link your program).

If your target machine comes with a C compiler, it probably comes with suitable header files also. If you make these files accessible from the host machine, the cross-compiler can use them also.

Otherwise, you're on your own in finding header files to use when cross-compiling.

When you have found suitable header files, put them in the directory '`/usr/local/`*target*`/include`', before building the cross compiler. Then installation will run fixincludes properly and install the corrected versions of the header files where the compiler will use them.

Provide the header files before you build the cross-compiler, because the build stage actually runs the cross-compiler to produce parts of '`libgcc.a`'. (These are the parts that *can* be compiled with GNU CC.) Some of them need suitable header files.

Here's an example showing how to copy the header files from a target machine. On the target machine, do this:

```
(cd /usr/include; tar cf - .) > tarfile
```

Then, on the host machine, do this:

```
ftp target-machine
lcd /usr/local/target/include
get tarfile
quit
tar xf tarfile
```

3.4.6 Actually Building the Cross-Compiler

Now you can proceed just as for compiling a single-machine compiler through the step of building stage 1. If you have not provided some sort of '`libgcc1.a`', then compilation will give up at the point where it needs that file, printing a suitable error message. If you do provide '`libgcc1.a`', then building the compiler will automatically compile and link a test program called '`libgcc1-test`'; if you get errors in the linking, it means that not all of the necessary routines in '`libgcc1.a`' are available.

You must provide the header file '`float.h`'. One way to do this is to compile '`enquire`' and run it on your target machine. The job of '`enquire`' is to run on the target machine and figure out by experiment the nature of its floating point representation. '`enquire`' records its findings in the header file '`float.h`'. If you can't produce this file by running '`enquire`' on the target machine, then you will need to come up with a suitable '`float.h`' in some other way (or else, avoid using it in your programs).

Do not try to build stage 2 for a cross-compiler. It doesn't work to rebuild GNU CC as a cross-compiler using the cross-compiler, because that would produce a program that runs on the target machine, not on the host. For example, if you compile a 386-to-68030 cross-compiler with itself, the result will not be right either for the 386 (because it was compiled into 68030 code) or for the 68030 (because it was configured for a 386 as the host). If you want to compile GNU CC into 68030 code, whether you compile it on a 68030 or with a cross-compiler on a 386, you must specify a 68030 as the host when you configure it.

To install the cross-compiler, use '`make install`', as usual.

3.5 Installing GNU CC on the Sun

On Solaris, do not use the linker or other tools in '`/usr/ucb`' to build GNU CC. Use `/usr/ccs/bin`.

If the assembler reports '`Error: misaligned data`' when bootstrapping, you are probably using an obsolete version of the GNU assembler. Upgrade to the latest version of GNU `binutils`, or use the Solaris assembler.

Make sure the environment variable `FLOAT_OPTION` is not set when you compile '`libgcc.a`'. If this option were set to `f68881` when '`libgcc.a`' is compiled, the resulting code would demand to be linked with a special startup file and would not link properly without special pains.

There is a bug in `alloca` in certain versions of the Sun library. To avoid this bug, install the binaries of GNU CC that were compiled by GNU CC. They use `alloca` as a built-in function and never the one in the library.

Some versions of the Sun compiler crash when compiling GNU CC. The problem is a segmentation fault in cpp. This problem seems to be due to the bulk of data in the environment variables. You may be able to avoid it by using the following command to compile GNU CC with Sun CC:

```
make CC="TERMCAP=x OBJS=x LIBFUNCS=x STAGESTUFF=x cc"
```

SunOS 4.1.3 and 4.1.3_U1 have bugs that can cause intermittent core dumps when compiling GNU CC. A common symptom is an internal compiler error which does not recur if you run it again. To fix the problem, install Sun recommended patch 100726 (for SunOS 4.1.3) or 101508 (for SunOS 4.1.3_U1), or upgrade to a later SunOS release.

3.6 Installing GNU CC on VMS

The VMS version of GNU CC is distributed in a backup saveset containing both source code and precompiled binaries.

To install the '`gcc`' command so you can use the compiler easily, in the same manner as you use the VMS C compiler, you must install the VMS CLD file for GNU CC as follows:

1. Define the VMS logical names '`GNU_CC`' and '`GNU_CC_INCLUDE`' to point to the directories where the GNU CC executables ('`gcc-cpp.exe`', '`gcc-cc1.exe`', etc.) and the C include files are kept respectively. This should be done with the commands:

   ```
   $ assign /system /translation=concealed -
     disk:[gcc.] gnu_cc
   $ assign /system /translation=concealed -
     disk:[gcc.include.] gnu_cc_include
   ```

   with the appropriate disk and directory names. These commands can be placed in your system startup file so they will be executed whenever the machine is rebooted. You may, if you choose, do this via the '`GCC_INSTALL.COM`' script in the '`[GCC]`' directory.

2. Install the '`GCC`' command with the command line:

   ```
   $ set command /table=sys$common:[syslib]dcltables -
     /output=sys$common:[syslib]dcltables gnu_cc:[000000]gcc
   $ install replace sys$common:[syslib]dcltables
   ```

3. To install the help file, do the following:

```
$ library/help sys$library:helplib.hlb gcc.hlp
```

Now you can invoke the compiler with a command like '`gcc /verbose file.c`', which is equivalent to the command '`gcc -v -c file.c`' in Unix.

If you wish to use GNU C++ you must first install GNU CC, and then perform the following steps:

1. Define the VMS logical name '`GNU_GXX_INCLUDE`' to point to the directory where the preprocessor will search for the C++ header files. This can be done with the command:

   ```
   $ assign /system /translation=concealed -
     disk:[gcc.gxx_include.] gnu_gxx_include
   ```

   with the appropriate disk and directory name. If you are going to be using a C++ runtime library, this is where its install procedure will install its header files.
2. Obtain the file '`gcc-cc1plus.exe`', and place this in the same directory that '`gcc-cc1.exe`' is kept.

   The GNU C++ compiler can be invoked with a command like '`gcc /plus /verbose file.cc`', which is equivalent to the command '`g++ -v -c file.cc`' in Unix.

We try to put corresponding binaries and sources on the VMS distribution tape. But sometimes the binaries will be from an older version than the sources, because we don't always have time to update them. (Use the '`/version`' option to determine the version number of the binaries and compare it with the source file '`version.c`' to tell whether this is so.) In this case, you should use the binaries you get to recompile the sources. If you must recompile, here is how:

1. Execute the command procedure '`vmsconfig.com`' to set up the files '`tm.h`', '`config.h`', '`aux-output.c`', and '`md.`', and to create files '`tconfig.h`' and '`hconfig.h`'. This procedure also creates several linker option files used by '`make-cc1.com`' and a data file used by '`make-l2.com`'.

   ```
   $ @vmsconfig.com
   ```
2. Setup the logical names and command tables as defined above. In addition, define the VMS logical name '`GNU_BISON`' to point at the to the directories where the Bison executable is kept. This should be done with the command:

   ```
   $ assign /system /translation=concealed -
     disk:[bison.] gnu_bison
   ```

   You may, if you choose, use the '`INSTALL_BISON.COM`' script in the '`[BISON]`' directory.
3. Install the '`BISON`' command with the command line:

   ```
   $ set command /table=sys$common:[syslib]dcltables -
     /output=sys$common:[syslib]dcltables -
     gnu_bison:[000000]bison
   $ install replace sys$common:[syslib]dcltables
   ```
4. Type '`@make-gcc`' to recompile everything (alternatively, submit the file '`make-gcc.com`' to a batch queue). If you wish to build the GNU C++ compiler as well as the GNU CC compiler, you must first edit '`make-gcc.com`' and follow the instructions that appear in the comments.
5. In order to use GCC, you need a library of functions which GCC compiled code will call to perform certain tasks, and these functions are defined in the file '`libgcc2.c`'.

To compile this you should use the command procedure '`make-l2.com`', which will generate the library '`libgcc2.olb`'. '`libgcc2.olb`' should be built using the compiler built from the same distribution that '`libgcc2.c`' came from, and '`make-gcc.com`' will automatically do all of this for you.

To install the library, use the following commands:

```
$ library gnu_cc:[000000]gcclib/delete=(new,eprintf)
$ library gnu_cc:[000000]gcclib/delete=L_*
$ library libgcc2/extract=*/output=libgcc2.obj
$ library gnu_cc:[000000]gcclib libgcc2.obj
```

The first command simply removes old modules that will be replaced with modules from '`libgcc2`' under different module names. The modules `new` and `eprintf` may not actually be present in your '`gcclib.olb`'—if the VMS librarian complains about those modules not being present, simply ignore the message and continue on with the next command. The second command removes the modules that came from the previous version of the library '`libgcc2.c`'.

Whenever you update the compiler on your system, you should also update the library with the above procedure.

6. You may wish to build GCC in such a way that no files are written to the directory where the source files reside. An example would be the when the source files are on a read-only disk. In these cases, execute the following DCL commands (substituting your actual path names):

```
$ assign dua0:[gcc.build_dir.]/translation=concealed, -
         dua1:[gcc.source_dir.]/translation=concealed  gcc_build
$ set default gcc_build:[000000]
```

   where the directory '`dua1:[gcc.source_dir]`' contains the source code, and the directory '`dua0:[gcc.build_dir]`' is meant to contain all of the generated object files and executables. Once you have done this, you can proceed building GCC as described above. (Keep in mind that '`gcc_build`' is a rooted logical name, and thus the device names in each element of the search list must be an actual physical device name rather than another rooted logical name).

7. **If you are building GNU CC with a previous version of GNU CC, you also should check to see that you have the newest version of the assembler**. In particular, GNU CC version 2 treats global constant variables slightly differently from GNU CC version 1, and GAS version 1.38.1 does not have the patches required to work with GCC version 2. If you use GAS 1.38.1, then `extern const` variables will not have the read-only bit set, and the linker will generate warning messages about mismatched psect attributes for these variables. These warning messages are merely a nuisance, and can safely be ignored.

   If you are compiling with a version of GNU CC older than 1.33, specify '`/DEFINE=("inline=")`' as an option in all the compilations. This requires editing all the `gcc` commands in '`make-cc1.com`'. (The older versions had problems supporting `inline`.) Once you have a working 1.33 or newer GNU CC, you can change this file back.

8. If you want to build GNU CC with the VAX C compiler, you will need to make minor changes in '`make-cccp.com`' and '`make-cc1.com`' to choose alternate definitions of `CC`, `CFLAGS`, and `LIBS`. See comments in those files. However, you must also

have a working version of the GNU assembler (GNU as, aka GAS) as it is used as the back-end for GNU CC to produce binary object modules and is not included in the GNU CC sources. GAS is also needed to compile 'libgcc2' in order to build 'gcclib' (see above); 'make-l2.com' expects to be able to find it operational in 'gnu_cc:[000000]gnu-as.exe'.

To use GNU CC on VMS, you need the VMS driver programs 'gcc.exe', 'gcc.com', and 'gcc.cld'. They are distributed with the VMS binaries ('gcc-vms') rather than the GNU CC sources. GAS is also included in 'gcc-vms', as is Bison.

Once you have successfully built GNU CC with VAX C, you should use the resulting compiler to rebuild itself. Before doing this, be sure to restore the `CC`, `CFLAGS`, and `LIBS` definitions in 'make-cccp.com' and 'make-cc1.com'. The second generation compiler will be able to take advantage of many optimizations that must be suppressed when building with other compilers.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked with the sharable 'VAXCRTL' library. Now this should work.

Even with this version, however, GNU CC itself should not be linked with the sharable 'VAXCRTL'. The version of `qsort` in 'VAXCRTL' has a bug (known to be present in VMS versions V4.6 through V5.5) which causes the compiler to fail.

The executables are generated by 'make-cc1.com' and 'make-cccp.com' use the object library version of 'VAXCRTL' in order to make use of the `qsort` routine in 'gcclib.olb'. If you wish to link the compiler executables with the shareable image version of 'VAXCRTL', you should edit the file 'tm.h' (created by 'vmsconfig.com') to define the macro `QSORT_WORKAROUND`.

`QSORT_WORKAROUND` is always defined when GNU CC is compiled with VAX C, to avoid a problem in case 'gcclib.olb' is not yet available.

3.7 collect2

GNU CC uses a utility called `collect2` on nearly all systems to arrange to call various initialization functions at start time.

The program `collect2` works by linking the program once and looking through the linker output file for symbols with particular names indicating they are constructor functions. If it finds any, it creates a new temporary '.c' file containing a table of them, compiles it, and links the program a second time including that file.

The actual calls to the constructors are carried out by a subroutine called `__main`, which is called (automatically) at the beginning of the body of `main` (provided `main` was compiled with GNU CC). Calling `__main` is necessary, even when compiling C code, to allow linking C and C++ object code together. (If you use '-nostdlib', you get an unresolved reference to `__main`, since it's defined in the standard GCC library. Include '-lgcc' at the end of your compiler command line to resolve this reference.)

The program `collect2` is installed as `ld` in the directory where the passes of the compiler are installed. When `collect2` needs to find the *real* `ld`, it tries the following file names:

- 'real-ld' in the directories listed in the compiler's search directories.
- 'real-ld' in the directories listed in the environment variable `PATH`.

- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- ‘`ld`’ in the compiler’s search directories, except that `collect2` will not execute itself recursively.
- ‘`ld`’ in `PATH`.

“The compiler’s search directories” means all the directories where `gcc` searches for passes of the compiler. This includes directories that you specify with ‘`-B`’.

Cross-compilers search a little differently:

- ‘`real-ld`’ in the compiler’s search directories.
- ‘*target*`-real-ld`’ in `PATH`.
- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- ‘`ld`’ in the compiler’s search directories.
- ‘*target*`-ld`’ in `PATH`.

`collect2` explicitly avoids running `ld` using the file name under which `collect2` itself was invoked. In fact, it remembers up a list of such names—in case one copy of `collect2` finds another copy (or version) of `collect2` installed as `ld` in a second place in the search path.

`collect2` searches for the utilities `nm` and `strip` using the same algorithm as above for `ld`.

3.8 Standard Header File Directories

`GCC_INCLUDE_DIR` means the same thing for native and cross. It is where GNU CC stores its private include files, and also where GNU CC stores the fixed include files. A cross compiled GNU CC runs `fixincludes` on the header files in ‘`$(tooldir)/include`’. (If the cross compilation header files need to be fixed, they must be installed before GNU CC is built. If the cross compilation header files are already suitable for ANSI C and GNU CC, nothing special need be done).

`GPLUSPLUS_INCLUDE_DIR` means the same thing for native and cross. It is where `g++` looks first for header files. The C++ library installs only target independent header files in that directory.

`LOCAL_INCLUDE_DIR` is used only for a native compiler. It is normally ‘`/usr/local/include`’. GNU CC searches this directory so that users can install header files in ‘`/usr/local/include`’.

`CROSS_INCLUDE_DIR` is used only for a cross compiler. GNU CC doesn’t install anything there.

`TOOL_INCLUDE_DIR` is used for both native and cross compilers. It is the place for other packages to install header files that GNU CC will use. For a cross-compiler, this is the equivalent of ‘`/usr/include`’. When you build a cross-compiler, `fixincludes` processes any header files in this directory.

4 Extensions to the C Language Family

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

These extensions are available in C and Objective C. Most of them are also available in C++. See Chapter 5 [Extensions to the C++ Language], page 177, for extensions that apply *only* to C++.

4.1 Statements and Declarations in Expressions

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

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

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

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

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

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

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

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

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

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

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see Section 4.7 [Typeof], page 142) or type naming (see Section 4.6 [Naming Types], page 142).

4.2 Locally Declared Labels

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

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, ...;
```

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

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

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

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

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

4.4 Nested Functions

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

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

  return square (a) + square (b);
}
```

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

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

[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.

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

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

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

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

    intermediate (store, size);
  }
```

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

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

GNU CC implements taking the address of a nested function using a technique called *trampolines*. A paper describing them is available as '`http://master.debian.org/~karlheg/Usenix88-lexic`

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see Section 4.2 [Local Labels], page 138). 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];
    }
  ...
}
```

4.5 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).

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

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

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

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

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

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

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

4.6 Naming an Expression's Type

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

```
typedef name = exp;
```

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

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

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

4.7 Referring to a Type with `typeof`

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

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

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

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

Here is an example with a typename as the argument:

```
typeof (int *)
```

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

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`. See Section 4.35 [Alternate Keywords], page 173.

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

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

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

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

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

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

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

  It is equivalent to the following traditional C declaration:

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

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

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

  Now the declaration can be rewritten this way:

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

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

4.8 Generalized Lvalues

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.9 Conditionals with Omitted Operands

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

Therefore, the expression

```
x ? : y
```

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

This example is perfectly equivalent to

```
x ? x : y
```

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

4.10 Double-Word Integers

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

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

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

4.11 Complex Numbers

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

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

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

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

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

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

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

4.12 Hex Floats

GNU CC recognizes 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. 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 significand 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`.

4.13 Arrays of Length Zero

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

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline = (struct line *)
    malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

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

4.14 Arrays of Variable Length

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

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

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

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

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

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

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

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

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

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

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

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

4.15 Macros with Variable Numbers of Arguments

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

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

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

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

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

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

```
fprintf (stderr, "success!\n" , )
```

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

```
fprintf (stderr, "success!\n")
```

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

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

4.16 Non-Lvalue Arrays May Have Subscripts

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

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

struct foo f();

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

4.17 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.

4.18 Non-Constant Initializers

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

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

4.19 Constructor Expressions

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

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

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

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

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

This is equivalent to writing the following:

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

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

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

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

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

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

4.20 Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In GNU C you can give the elements in any order, specifying the array indices or structure field names they apply to. This extension is not implemented in GNU C++.

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

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

is equivalent to

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

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

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

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

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

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

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

the following initialization

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

is equivalent to

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

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

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

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

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

union foo f = { d: 4 };
```

will convert 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer `i`, since it is an integer. (See Section 4.22 [Cast to Union], page 151.)

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

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

is equivalent to

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

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

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

4.21 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:
```

4.22 Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with `union` *tag* or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts. (See Section 4.19 [Constructors], page 149.)

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);
```

4.23 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. Nine attributes, `noreturn`, `const`, `format`, `no_instrument_function`, `section`, `constructor`, `destructor`, `unused` and `weak` are currently defined for functions. Other attributes, including `section` are supported for variables declarations (see Section 4.29 [Variable Attributes], page 158) and for types (see Section 4.30 [Type Attributes], page 161).

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

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

```
void fatal () __attribute__ ((noreturn));

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

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

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

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

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

```
typedef void voidfn ();

volatile voidfn fatal;
```

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

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

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

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

```
typedef int intfn ();

extern const intfn square;
```

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

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

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

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

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

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

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

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

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

`format_arg (`*string-index*`)`

The `format_arg` attribute specifies that a function takes `printf` or `scanf` style arguments, modifies it (for example, to translate it into another language), and passes it to a `printf` or `scanf` style function. 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 `my_dgettext` whose result is passed to a `printf`, `scanf`, or `strftime` type function for consistency with the `printf` style format string argument `my_format`.

The parameter *string-index* specifies which argument is the format string argument (starting from 1).

The `format-arg` attribute allows you to identify your own functions which modify format strings, so that GNU CC can check the calls to `printf`, `scanf`, or `strftime` function whose operands are a call to one of your own function. The compiler always treats `gettext`, `dgettext`, and `dcgettext` in this manner.

`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. GNU CC will not produce a warning for this function. GNU C++ does not currently support this attribute as definitions without parameters are valid in C++.

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

`alias ("target")`
: The `alias` attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () { /* do something */; }
void f () __attribute__ ((weak, alias ("__f")));
```

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

Not all target machines support this attribute.

`no_check_memory_usage`
: If '`-fcheck-memory-usage`' is given, calls to support routines will be generated before most memory accesses, to permit support code to record usage and detect uses of uninitialized or unallocated storage. Since the compiler cannot handle them properly, `asm` statements are not allowed. Declaring a function with this attribute disables the memory checking code for that function, permitting the use of `asm` statements without requiring separate compilation with different options, and allowing you to write support routines of your own if you wish, without getting infinite recursion if they get compiled with this option.

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

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

  The PowerPC compiler for Windows NT currently ignores the `stdcall` attribute.

`cdecl`
: On the Intel 386, the `cdecl` attribute causes the compiler to assume that the calling function will pop off the stack space used to pass arguments. This is useful to override the effects of the '`-mrtd`' switch.

  The PowerPC compiler for Windows NT currently ignores the `cdecl` attribute.

`longcall`
: On the RS/6000 and PowerPC, the `longcall` attribute causes the compiler to always call the function via a pointer, so that functions which reside further than 64 megabytes (67,108,864 bytes) from the current location can be called.

`dllimport`
: On the PowerPC running Windows NT, the `dllimport` attribute causes the compiler to call the function via a global pointer to the function pointer that is set up by the Windows NT dll library. The pointer name is formed by combining `__imp_` and the function name.

`dllexport`
: On the PowerPC running Windows NT, the `dllexport` attribute causes the compiler to provide a global pointer to the function pointer, so that it can be called with the `dllimport` attribute. The pointer name is formed by combining `__imp_` and the function name.

`exception (`*except-func* `[,` *except-arg*`])`
: On the PowerPC running Windows NT, the `exception` attribute causes the compiler to modify the structured exception table entry it emits for the declared function. The string or identifier *except-func* is placed in the third entry of the structured exception table. It represents a function, which is called by the exception handling mechanism if an exception occurs. If it was specified, the string or identifier *except-arg* is placed in the fourth entry of the structured exception table.

`function_vector`
: Use this option on the H8/300 and H8/300H to indicate that the specified function should be called through the function vector. Calling a function through the function vector will reduce code size, however; the function vector has a limited size (maximum 128 entries on the H8/300 and 64 entries on the H8/300H) and shares space with the interrupt vector.

  You must use GAS and GLD from GNU binutils version 2.7 or later for this option to work correctly.

`interrupt_handler`
: Use this option on the H8/300 and H8/300H 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.

`eightbit_data`
: Use this option on the H8/300 and H8/300H to indicate that the specified variable should be placed into the eight bit data section. The compiler will generate more efficient code for certain operations on data in the eight bit data area. Note the eight bit data area is limited to 256 bytes of data.

  You must use GAS and GLD from GNU binutils version 2.7 or later for this option to work correctly.

`tiny_data`
: Use this option on the H8/300H to indicate that the specified variable should be placed into the tiny data section. The compiler will generate more efficient code for loads and stores on data in the tiny data section. Note the tiny data area is limited to slightly under 32kbytes of data.

`interrupt`
: Use this option on the M32R/D 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.

`model (`*model-name*`)`
: Use this attribute on the M32R/D to set the addressability of an object, and the code generated for a function. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

  Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and are callable with the `bl` instruction.

  Medium model objects may live anywhere in the 32 bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and are callable with the `bl` instruction.

  Large model objects may live anywhere in the 32 bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and may not be reachable with the `bl` instruction (the compiler will generate the much slower `seth/add3/jl` instruction sequence).

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

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

1. It is impossible to generate `#pragma` commands from a macro.
2. There is no telling what the same `#pragma` might mean in another compiler.

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

4.24 Prototypes and Old-Style Function Definitions

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

```
/* Use prototypes unless the compiler is old-fashioned.  */
#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`. ANSI C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an `int`, which does not match the prototype argument type of `short`.

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

```
int isroot (uid_t);

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

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

4.25 C++ Style Comments

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

4.26 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.

4.27 The Character ⟨ESC⟩ in Constants

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

4.28 Inquiring on Alignment of Types or Variables

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

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

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

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

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

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

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

4.29 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Eight attributes are currently defined for variables: `aligned`, `mode`, `nocommon`, `packed`, `section`, `transparent_union`, `unused`, and `weak`. Other attributes are available for functions (see Section 4.23 [Function Attributes], page 151) and for types (see Section 4.30 [Type Attributes], page 161).

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

`aligned` (*alignment*)

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

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

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

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

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

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`mode` (*mode*)

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of '`byte`' or '`__byte__`' to indicate the mode corresponding to a one-byte integer, '`word`' or '`__word__`' for the mode of a one-word integer, and '`pointer`' or '`__pointer__`' for the mode used to represent pointers.

`nocommon` This attribute specifies requests GNU CC not to place a variable "common" but instead to allocate space for it directly. If you specify the '`-fno-common`' flag, GNU CC will do this for all variables.

Specifying the `nocommon` attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

`packed` The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

`section ("section-name")`

Normally, the compiler places the objects it generates in sections like `data` and `bss`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `section` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __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. GNU CC issues a warning and otherwise ignores the `section` attribute in uninitialized variable declarations.

You may only use the `section` attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the `common` (or `bss`) section and can be multiply "defined". You can force a variable to be initialized with the '`-fno-common`' flag or the `nocommon` attribute.

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

`transparent_union`
: This attribute, attached to a function parameter which is a union, means that the corresponding argument may have the type of any union member, but the argument is passed as if its type were that of the first union member. For more details see See Section 4.30 [Type Attributes], page 161. 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. GNU CC will not produce a warning for this variable.

`weak`
: The `weak` attribute is described in See Section 4.23 [Function Attributes], page 151.

`model (`*model-name*`)`
: Use this attribute on the M32R/D to set the addressability of an object. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction).

Medium and large model objects may live anywhere in the 32 bit address space (the compiler will generate `seth/add3` instructions to load their addresses).

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

4.30 Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of `struct` and `union` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Three attributes are currently defined for types: `aligned`, `packed`, and `transparent_union`. Other attributes are defined for functions (see Section 4.23 [Function Attributes], page 151) and for variables (see Section 4.29 [Variable Attributes], page 158).

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.

`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 ANSI C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

`packed` This attribute, attached to an `enum`, `struct`, or `union` type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for `struct` and `union` types is equivalent to specifying the `packed` attribute on each of the structure or union members. Specifying the '`-fshort-enums`' flag on the line is equivalent to specifying the `packed` attribute on all `enum` definitions.

You may only specify this attribute after a closing curly brace on an `enum` definition, not in a `typedef` declaration, unless that declaration also contains the definition of the `enum`.

`transparent_union`

This attribute, attached to a `union` type definition, indicates that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like `const` on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the `wait` function must accept either a value of type `int *` to comply with Posix, or a value of type `union wait *` to comply with the 4.1BSD interface. If `wait`'s parameter were `void *`, `wait` would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, `<sys/wait.h>` might define the interface as follows:

```
typedef union
  {
          int *__ip;
          union wait *__up;
        } wait_status_ptr_t __attribute__ ((__transparent_union__));

      pid_t wait (wait_status_ptr_t);
```

This interface allows either `int *` or `union wait *` arguments to be passed, using the `int *` calling convention. The program can call `wait` with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, `wait`'s implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
  return waitpid (-1, p.__ip, 0);
}
```

`unused` When attached to a type (including a `union` or a `struct`), this attribute means that variables of that type are meant to appear possibly unused. GNU CC 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.

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

4.31 An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation. If you don't use '`-O`', no function is really inline.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline__` instead of `inline`. See Section 4.35 [Alternate Keywords], page 173.) You can also make all "simple enough" functions inline with the option '`-finline-functions`'.

Note that certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of varargs, use of alloca, use of variable sized data types (see Section 4.14 [Variable Length], page 146), use of computed goto (see Section 4.3 [Labels as Values], page 138), use of nonlocal goto, and nested functions (see Section 4.4 [Nested Functions], page 139). 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.

GNU CC automatically inlines member functions defined within the class body of C++ programs even if they are not explicitly declared `inline`. (You can override this with '`-fno-default-inline`'; see Section 2.5 [Options Controlling C++ Dialect], page 16.)

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

GNU C does not inline any functions when not optimizing. It is not clear whether it is better to inline or not, in this case, but we found that a correct implementation when not optimizing was difficult. So we did the easy thing, and turned it off.

4.32 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can specify the operands of the instruction using C expressions. This means you need not guess which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
     asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has '`"f"`' as its operand constraint, saying that a floating point register is required. The '`=`' in '`=f`' indicates that the operand is an output; all output operands' constraints must use '`=`'. The constraints use the same language used in the machine description (see Section 16.6 [Constraints], page 293).

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 limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means or 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, GNU CC 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; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm supports input-output or read-write operands. Use the constraint character '`+`' to indicate such an operand and list it with the output operands.

When the constraints for the read-write operand (or the operand in which only some of the bits are to be changed) allows a register, you may, as an alternative, logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) '`combine`' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
     asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint '`"0"`' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work 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; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the VAX:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

It is an error for a clobber description to overlap an input or output operand (for example, an operand describing a register class with one member, mentioned in the clobber list). Most notably, it is invalid to describe that an input operand is modified, but unused as output. It has to be specified as an input and output operand anyway. Note that if there are only unused output operands, you will then also need to specify `volatile` for the `asm` construct, as described below.

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. GNU CC on some machines represents the condition codes as a specific hardware register; '`cc`' serves to name this register. On other machines, the condition code is handled differently, and specifying '`cc`' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add '`memory`' to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single `asm` template, separated either with newlines (written as '`\n`') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and most Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes the subroutine `_foo` accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '`&`' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '`&`' for each output operand that may not overlap an input. See Section 16.6.4 [Modifiers], page 298.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the `asm` construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one `asm` to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable `__arg` is used to make sure that the instruction operates on a proper `double` value, and to accept only those arguments `x` which can convert automatically to a `double`.

Another way to make sure the instruction operates on the correct data type is to use a cast in the `asm`. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would warn about using a pointer unless the caller explicitly casts it.

If an `asm` has output operands, GNU CC assumes for optimization purposes 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, GNU CC will know the instruction has side-effects and will not delete the instruction or move it outside of loops. If the side-effects of your instruction are not purely external, but will affect variables in your program in ways other than reading the inputs and clobbering the specified registers or memory, you should write the `volatile` keyword to prevent future versions of GNU CC from moving the instruction around within a core region.

An `asm` instruction without any operands or clobbers (and "old style" `asm`) will not be deleted or moved significantly, regardless, unless it is unreachable, the same wasy as if you had written a `volatile` keyword.

Note that even a volatile `asm` instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile `asm` instructions to remain perfectly consecutive. If you want consecutive output, use a single `asm`.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when 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.

If you are writing a header file that should be includable in ANSI C programs, write `__asm__` instead of `asm`. See Section 4.35 [Alternate Keywords], page 173.

4.32.1 i386 floating point asm operands

There are several rules on the usage of stack-like regs in asm_operands insns. These rules apply only to the operands that are stack-like regs:

1. Given a set of input regs that die in an asm_operands, it is necessary to know which are implicitly popped by the asm, and which must be explicitly popped by gcc.

   An input reg that is implicitly popped by the asm must be explicitly clobbered, unless it is constrained to match an output operand.
2. For any input reg that is implicitly popped by an asm, it is necessary to know how to adjust the stack to compensate for the pop. If any non-popped input is closer to the top of the reg-stack than the implicitly popped reg, it would not be possible to know what the stack looked like — it's not clear how the rest of the stack "slides up".

   All implicitly popped input regs must be closer to the top of the reg-stack than any input that is not implicitly popped.

   It is possible that if an input dies in an insn, reload might use the input reg for an output reload. Consider this example:

   ```
   asm ("foo" : "=t" (a) : "f" (b));
   ```

   This asm says that input B is not popped by the asm, and that the asm pushes a result onto the reg-stack, ie, 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)");
```

4.33 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.

4.33.1 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:

'`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.

'd', 'a', 'f', . . .
Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. 'd', 'a' and 'f' are defined on the 68000/68020 to stand for data, address and floating point registers.

'i' An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

'n' An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use 'n' rather than 'i'.

'I', 'J', 'K', . . . 'P'
Other letters in the range 'I' through 'P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, 'I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

'E' An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

'F' An immediate floating operand (expression code `const_double`) is allowed.

'G', 'H' 'G' and 'H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

's' An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use 's' instead of 'i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between -128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a 'moveq' instruction. We arrange for this to happen by defining the letter 'K' to mean "any integer outside the range -128 to 127", and then specifying 'Ks' in the operand constraints.

'`g`' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

'`X`' Any operand whatsoever is allowed.

'`0`', '`1`', '`2`', ... '`9`'
An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This 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.

'`Q`', '`R`', '`S`', ... '`U`'
Letters in the range '`Q`' through '`U`' may be defined in a machine-dependent fashion to stand for arbitrary operand types.

4.33.2 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.

4.33.3 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.

'`&`' Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

'`&`' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires '`&`' while others do not. See, for example, the '`movdf`' insn of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the '`mulsi3`' insn of the ARM.

'`&`' does not obviate the need to write '`=`'.

'`%`' Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints.

'`#`' Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

4.33.4 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are '`m`' and '`r`' (for memory and general-purpose registers respectively; see Section 16.6.1 [Simple Constraints], page 293), and '`I`', usually the letter indicating the most common immediate-constant format.

For each machine architecture, the '`config/`*machine*`.h`' file defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly interesting for `asm`. The constraints are defined through these macros:

`REG_CLASS_FROM_LETTER`
: Register class constraints (usually lower case).

`CONST_OK_FOR_LETTER_P`
: Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually upper case).

`CONST_DOUBLE_OK_FOR_LETTER_P`
: Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually upper case).

`EXTRA_CONSTRAINT`
: Special cases of registers or memory. This macro is not required, and is only defined for some machines.

Inspecting these macro definitions in the compiler source for your machine is the best way to be certain you have the right constraints. However, here is a summary of the machine-dependent constraints available on some particular machines.

ARM family—'`arm.h`'

`f`	Floating-point register
`F`	One of the floating-point constants 0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0 or 10.0
`G`	Floating-point constant that would satisfy the constraint '`F`' if it were negated
`I`	Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2
`J`	Integer in the range -4095 to 4095
`K`	Integer that satisfies constraint '`I`' when inverted (ones complement)
`L`	Integer that satisfies constraint '`I`' when negated (twos complement)
`M`	Integer in the range 0 to 32
`Q`	A memory reference where the exact address is in a single register ("`m`" is preferable for `asm` statements)
`R`	An item in the constant pool
`S`	A symbol in the text segment of the current file

AMD 29000 family—'`a29k.h`'

`l`	Local register 0

`b`	Byte Pointer (‘`BP`’) register
`q`	‘`Q`’ register
`h`	Special purpose register
`A`	First accumulator register
`a`	Other accumulator register
`f`	Floating point register
`I`	Constant greater than 0, less than 0x100
`J`	Constant greater than 0, less than 0x10000
`K`	Constant whose high 24 bits are on (1)
`L`	16 bit constant whose high 8 bits are on (1)
`M`	32 bit constant whose high 16 bits are on (1)
`N`	32 bit negative constant that fits in 8 bits
`O`	The constant 0x80000000 or, on the 29050, any 32 bit constant whose low 16 bits are 0.
`P`	16 bit negative constant that fits in 8 bits
`G` `H`	A floating point constant (in `asm` statements, use the machine independent ‘`E`’ or ‘`F`’ instead)

IBM RS6000—‘`rs6000.h`’

`b`	Address base register
`f`	Floating point register
`h`	‘`MQ`’, ‘`CTR`’, or ‘`LINK`’ register
`q`	‘`MQ`’ register
`c`	‘`CTR`’ register
`l`	‘`LINK`’ register
`x`	‘`CR`’ register (condition register) number 0
`y`	‘`CR`’ register (condition register)
`z`	‘`FPMEM`’ stack memory for FPR-GPR transfers
`I`	Signed 16 bit constant
`J`	Constant whose low 16 bits are 0
`K`	Constant whose high 16 bits are 0
`L`	Constant suitable as a mask operand
`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
`U`	System V Release 4 small data area reference

Intel 386—'`i386.h`'

`q`	'`a`', `b`, `c`, or `d` register
`A`	'`a`', or `d` register (for 64-bit ints)
`f`	Floating point register
`t`	First (top of stack) floating point register
`u`	Second floating point register
`a`	'`a`' register
`b`	'`b`' register
`c`	'`c`' register
`d`	'`d`' register
`D`	'`di`' register
`S`	'`si`' register
`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)
`G`	Standard 80387 floating point constant

Intel 960—'`i960.h`'

`f`	Floating point register (`fp0` to `fp3`)
`l`	Local register (`r0` to `r15`)
`b`	Global register (`g0` to `g15`)
`d`	Any local or global register
`I`	Integers from 0 to 31

`J`	0
`K`	Integers from -31 to 0
`G`	Floating point 0
`H`	Floating point 1

MIPS—'`mips.h`'

`d`	General-purpose integer register
`f`	Floating-point register (if available)
`h`	'`Hi`' register
`l`	'`Lo`' register
`x`	'`Hi`' or '`Lo`' register
`y`	General-purpose integer register
`z`	Floating-point status register
`I`	Signed 16 bit constant (for arithmetic instructions)
`J`	Zero
`K`	Zero-extended 16-bit constant (for logic instructions)
`L`	Constant with low 16 bits zero (can be loaded with `lui`)
`M`	32 bit constant which requires two instructions to load (a constant which is not '`I`', '`K`', or '`L`')
`N`	Negative 16 bit constant
`O`	Exact power of two
`P`	Positive 16 bit constant
`G`	Floating point zero
`Q`	Memory reference that can be loaded with more than one instruction ('`m`' is preferable for `asm` statements)
`R`	Memory reference that can be loaded with one instruction ('`m`' is preferable for `asm` statements)
`S`	Memory reference in external OSF/rose PIC format ('`m`' is preferable for `asm` statements)

Motorola 680x0—'`m68k.h`'

`a`	Address register
`d`	Data register
`f`	68881 floating-point register, if available
`x`	Sun FPA (floating-point) register, if available
`y`	First 16 Sun FPA registers, if available

`I`	Integer in the range 1 to 8
`J`	16 bit signed number
`K`	Signed number whose magnitude is greater than 0x80
`L`	Integer in the range -8 to -1
`M`	Signed number whose magnitude is greater than 0x100
`G`	Floating point constant that is not a 68881 constant
`H`	Floating point constant that can be used by Sun FPA

SPARC—'`sparc.h`'

`f`	Floating-point register that can hold 32 or 64 bit values.
`e`	Floating-point register that can hold 64 or 128 bit values.
`I`	Signed 13 bit constant
`J`	Zero
`K`	32 bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)
`G`	Floating-point zero
`H`	Signed 13 bit constant, sign-extended to 32 or 64 bits
`Q`	Memory reference that can be loaded with one instruction ('`m`' is more appropriate for `asm` statements)
`S`	Constant, or memory address
`T`	Memory address aligned to an 8-byte boundary
`U`	Even register

4.34 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
     extern func () asm ("FUNC");

     func (x, y)
          int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

4.35 Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses. Stores into local register variables may be deleted when they appear to be dead according to dataflow analysis. References to local register variables may be deleted or moved or simplified.

  These local variables are sometimes convenient for use with the extended `asm` feature (see Section 4.32 [Extended Asm], page 165), 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`.)

4.35.1 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.

4.35.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see Section 4.32 [Extended Asm], page 165). 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 GNU CC 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.

4.36 Alternate Keywords

The option '`-traditional`' disables certain keywords; '`-ansi`' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords `asm`, `typeof` and `inline` cannot be used since they won't work in a program compiled with '`-ansi`', while the keywords `const`, `volatile`, `signed`, `typeof` and `inline` won't work in a program compiled with '`-traditional`'.

The way to solve these problems is to put '`__`' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, `__const__` instead of `const`, and `__inline__` instead of `inline`.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'`-pedantic`' causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing `__extension__` before the expression. `__extension__` has no effect aside from this.

4.37 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++.

4.38 Function Names as Strings

GNU CC predefines two string variables to be the name of the current function. The variable `__FUNCTION__` is the name of the function as it appears in the source. The variable `__PRETTY_FUNCTION__` is the name of the function pretty printed in a language specific fashion.

These names are always the same in a C function, but in a C++ function they may be different. For example, this program:

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

class a {
 public:
  sub (int i)
    {
      printf ("__FUNCTION__ = %s\n", __FUNCTION__);
      printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
    }
};

int
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = int  a::sub (int)
```

These names are not macros: they are predefined string variables. For example, '`#ifdef __FUNCTION__`' does not have any special meaning inside a function, since the preprocessor does not do anything special with the identifier `__FUNCTION__`.

4.39 Getting the Return or Frame Address of a Function

These functions may be used to get information about the callers of a function.

`__builtin_return_address` (*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.

  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`.

  This function should only be used with a non-zero argument for debugging purposes.

`__builtin_frame_address` (*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.

  The caveats that apply to `__builtin_return_address` apply to this function as well.

4.40 Other built-in functions provided by GNU CC

GNU CC 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.

GNU CC includes builtin versions of many of the functions in the standard C library. These will always be treated as having the same meaning as the C library function even if you specify the '`-fno-builtin`' (see Section 2.4 [C Dialect Options], page 12) option. These functions correspond to the C library functions `alloca`, `ffs`, `abs`, `fabsf`, `fabs`, `fabsl`, `labs`, `memcpy`, `memcmp`, `strcmp`, `strcpy`, `strlen`, `sqrtf`, `sqrt`, `sqrtl`, `sinf`, `sin`, `sinl`, `cosf`, `cos`, and `cosl`.

You can use the builtin function `__builtin_constant_p` to determine if a value is known to be constant at compile-time and hence that GNU CC 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 GNU CC 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 builtin 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 builtin, GNU CC will never return 1 when you call the inline function with a string constant or constructor expression (see Section 4.19 [Constructors], page 149) and will not return 1 when you pass a constant numeric value to the inline function unless you specify the '`-O`' option.

4.41 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 superceded 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:

`-fthis-is-variable`
: In early versions of C++, assignment to this could be used to implement application-defined memory allocation. Now, allocation functions ('`operator new`') are the standard-conforming way to achieve the same effect.

`-fexternal-templates`
`-falt-external-templates`
: These are two of the many ways for g++ to implement template instantiation. See Section 5.5 [Template Instantiation], page 180. 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.

5 Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro `__GNUC__`. You can also use `__GNUG__` to test specifically for GNU C++ (see section "Standard Predefined Macros" in *The C Preprocessor*).

5.1 Named Return Values in C++

GNU C++ extends the function-definition syntax to allow you to specify a name for the result of a function outside the body of the definition, in C++ programs:

```
type
functionname (args) return resultname;
{
  ...
  body
  ...
}
```

You can use this feature to avoid an extra constructor call when a function result has a class type. For example, consider a function `m`, declared as '`X v = m ();`', whose result is of class `X`:

```
X
m ()
{
  X b;
  b.a = 23;
  return b;
}
```

Although `m` appears to have no arguments, in fact it has one implicit argument: the address of the return value. At invocation, the address of enough space to hold `v` is sent in as the implicit argument. Then `b` is constructed and its `a` field is set to the value 23. Finally, a copy constructor (a constructor of the form '`X(X&)`') is applied to `b`, with the (implicit) return value location as the target, so that `v` is now bound to the return value.

But this is wasteful. The local `b` is declared just to hold something that will be copied right out. While a compiler that combined an "elision" algorithm with interprocedural data flow analysis could conceivably eliminate all of this, it is much more practical to allow you to assist the compiler in generating efficient code by manipulating the return value explicitly, thus avoiding the local variable and copy constructor altogether.

Using the extended GNU C++ function-definition syntax, you can avoid the temporary allocation and copying by naming `r` as your return value at the outset, and assigning to its `a` field directly:

```
X
m () return r;
{
        r.a = 23;
      }
```

The declaration of `r` is a standard, proper declaration, whose effects are executed **before** any of the body of `m`.

Functions of this type impose no additional restrictions; in particular, you can execute `return` statements, or return implicitly by reaching the end of the function body ("falling off the edge"). Cases like

```
      X
      m () return r (23);
      {
        return;
      }
```

(or even '`X m () return r (23); { }`') are unambiguous, since the return value `r` has been initialized in either case. The following code may be hard to read, but also works predictably:

```
      X
      m () return r;
      {
        X b;
        return b;
      }
```

The return value slot denoted by `r` is initialized at the outset, but the statement '`return b;`' overrides this value. The compiler deals with this by destroying `r` (calling the destructor if there is one, or doing nothing if there is not), and then reinitializing `r` with `b`.

This extension is provided primarily to help people who use overloaded operators, where there is a great need to control not just the arguments, but the return values of functions. For classes where the copy constructor incurs a heavy performance penalty (especially in the common case where there is a quick default constructor), this is a major savings. The disadvantage of this extension is that you do not control when the default constructor for the return value is called: it is always called at the beginning.

5.2 Minimum and Maximum Operators in C++

It is very convenient to have operators which return the "minimum" or the "maximum" of two arguments. In GNU C++ (but not in GNU C),

a `<?` *b* is the *minimum*, returning the smaller of the numeric values *a* and *b*;

a `>?` *b* is the *maximum*, returning the larger of the numeric values *a* and *b*.

These operations are not primitive in ordinary C++, since you can use a macro to return the minimum of two things in C++, as in the following example.

```
      #define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

You might then use '`int min = MIN (i, j);`' to set *min* to the minimum value of variables *i* and *j*.

However, side effects in `X` or `Y` may cause unintended behavior. For example, `MIN (i++, j++)` will fail, incrementing the smaller counter twice. A GNU C extension allows you to write safe macros that avoid this kind of problem (see Section 4.6 [Naming an Expression's

Type], page 142). 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.

5.3 `goto` and Destructors in GNU C++

In C++ programs, you can safely use the `goto` statement. When you use it to exit a block which contains aggregates requiring destructors, the destructors will run before the `goto` transfers control.

The compiler still forbids using `goto` to *enter* a scope that requires constructors.

5.4 Declarations and Definitions in One Header

C++ object definitions can be quite complex. In principle, your source code will need two kinds of things for each object that you use across more than one source file. First, you need an *interface* specification, describing its structure with type declarations and function prototypes. Second, you need the *implementation* itself. It can be tedious to maintain a separate interface description in a header file, in parallel to the actual implementation. It is also dangerous, since separate interface and implementation definitions may not remain parallel.

With GNU C++, you can use a single header file for both purposes.

> *Warning:* The mechanism to specify this is in transition. For the nonce, you must use one of two `#pragma` commands; in a future release of GNU C++, an alternative mechanism will make these `#pragma` commands unnecessary.

The header file contains the full definitions, but is marked with '`#pragma interface`' in the source code. This allows the compiler to use the header file only as an interface specification when ordinary source files incorporate it with `#include`. In the single source file where the full implementation belongs, you can use either a naming convention or '`#pragma implementation`' to indicate this alternate use of the header file.

`#pragma interface`
`#pragma interface "`*subdir*`/`*objects*`.h"`

> Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing '`#pragma interface`' is included in a compilation, this auxiliary information will not be generated (unless the main input source file itself uses '`#pragma implementation`'). Instead, the object files will contain references to be resolved at link time.
>
> The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to '`#pragma implementation`'.

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

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use '`#pragma interface`'. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use '`#pragma implementation`' with no argument, it applies to an include file with the same basename[1] as your source file. For example, in '`allclass.cc`', giving just '`#pragma implementation`' by itself is equivalent to '`#pragma implementation "allclass.h"`'.

In versions of GNU C++ prior to 2.6.0 '`allclass.h`' was treated as an implementation file whenever you would include it from '`allclass.cc`' even if you never specified '`#pragma implementation`'. This was deemed to be more trouble than it was worth, however, and disabled.

If you use an explicit '`#pragma implementation`', it must appear in your source file *before* you include the affected header files.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use '`#include`' to include the header file; '`#pragma implementation`' only specifies how to use the file—it doesn't actually include it.)

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

'`#pragma implementation`' and '`#pragma interface`' also have an effect on function inlining.

If you define a class in a header file marked with '`#pragma interface`', the effect on a function defined in that class is similar to an explicit `extern` declaration—the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as '`#pragma implementation`', the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without 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.

5.5 Where's the Template?

C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system. Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which I will refer to as the Borland model and the Cfront model.

[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`'.

Borland model

Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; the compiler emits template instances in each translation unit that uses them, and the linker collapses them together. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. This disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all templates in the header file, since they must be seen to be instantiated.

Cfront model

The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. A more modern version of the repository works as follows: As individual object files are built, the compiler places any template definitions and instantiations encountered in the repository. At link time, the link wrapper adds in the objects in the repository and compiles any needed instances that were not previously emitted. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; for some code this can be just as transparent, but in practice it can been very difficult to build multiple programs in one directory and one program in multiple directories. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which should be compiled separately.

When used with GNU ld version 2.8 or later on an ELF system such as Linux/GNU or Solaris 2, or on Microsoft Windows, g++ supports the Borland model. On other systems, g++ implements neither automatic model.

A future version of g++ will support a hybrid model whereby the compiler will emit any instantiations for which the template definition is included in the compile, and store template definitions and instantiation context information into the object file for the rest. The link wrapper will extract that information as necessary and invoke the compiler to produce the remaining instantiations. The linker will then combine duplicate instantiations.

In the mean time, you have the following options for dealing with template instantiations:

1. Compile your template-using code with '`-frepo`'. The compiler will generate files with the extension '`.rpo`' listing all of the template instantiations used in the corresponding object files which could be instantiated there; the link wrapper, '`collect2`', will then update the '`.rpo`' files to tell the compiler where to place those instantiations and rebuild any affected object files. The link-time overhead is negligible after the first pass, as the compiler will continue to place the instantiations in the same files.

   This is your best option for application code written for the Borland model, as it will just work. Code written for the Cfront model will need to be modified so that the template definitions are available at one or more points of instantiation; usually this is as simple as adding `#include <tmethods.cc>` to the end of each template header.

For library code, if you want the library to provide all of the template instantiations it needs, just try to link all of its object files together; the link will fail, but cause the instantiations to be generated as a side effect. Be warned, however, that this may cause conflicts if multiple libraries try to provide the same instantiations. For greater control, use explicit instantiation as described in the next option.

2. Compile your code with '`-fno-implicit-templates`' to disable the implicit generation of template instances, and explicitly instantiate all the ones you use. This approach requires more knowledge of exactly which instances you need than do the others, but it's less mysterious and allows greater control. You can scatter the explicit instantiations throughout your program, perhaps putting them in the translation units where the instances are used or the translation units that define the templates themselves; you can put all of the explicit instantiations you need into one big file; or you can create small files like

   ```
   #include "Foo.h"
   #include "Foo.cc"

   template class Foo<int>;
   template ostream& operator <<
                   (ostream&, const Foo<int>&);
   ```

   for each of the instances you need, and create a template instantiation library from those.

   If you are using Cfront-model code, you can probably get away with not using '`-fno-implicit-templates`' when compiling files that don't '`#include`' the member template definitions.

   If you use one big file to do the instantiations, you may want to compile it without '`-fno-implicit-templates`' so you get all of the instances required by your explicit instantiations (but not by any other files) without having to specify them as well.

   g++ has extended the template instantiation syntax outlined in the Working Paper to allow forward declaration of explicit instantiations and instantiation of the compiler support data for a template class (i.e. the vtable) without instantiating any of its members:

   ```
   extern template int max (int, int);
   inline template class Foo<int>;
   ```

3. Do nothing. Pretend g++ does implement automatic instantiation management. Code written for the Borland model will work fine, but each translation unit will contain instances of each of the templates it uses. In a large program, this can lead to an unacceptable amount of code duplication.

4. Add '`#pragma interface`' to all files containing template definitions. For each of these files, add '`#pragma implementation "`*filename*`"`' to the top of some '`.C`' file which '`#include`'s it. Then compile everything with '`-fexternal-templates`'. The templates will then only be expanded in the translation unit which implements them (i.e. has a '`#pragma implementation`' line for the file where they live); all other files will use external references. If you're lucky, everything should work properly. If you get undefined symbol errors, you need to make sure that each template instance which is used in the program is used in the file which implements that template. If you don't

have any use for a particular instance in that file, you can just instantiate it explicitly, using the syntax from the latest C++ working paper:

```
    template class A<int>;
    template ostream& operator << (ostream&, const A<int>&);
```

This strategy will work with code written for either model. If you are using code written for the Cfront model, the file containing a class template and the file containing its member templates should be implemented in the same translation unit.

A slight variation on this approach is to instead use the flag '`-falt-external-templates`'; this flag causes template instances to be emitted in the translation unit that implements the header where they are first instantiated, rather than the one which implements the file where the templates are defined. This header must be the same in all translation units, or things are likely to break.

See Section 5.4 [Declarations and Definitions in One Header], page 179, for more discussion of these pragmas.

5.6 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);
```

You must specify '`-Wno-pmf-conversions`' to use this extension.

5.7 Type Abstraction using Signatures

In GNU C++, you can use the keyword `signature` to define a completely abstract class interface as a datatype. You can connect this abstraction with actual classes using signature pointers. If you want to use signatures, run the GNU compiler with the '`-fhandle-signatures`' command-line option. (With this option, the compiler reserves a second keyword `sigof` as well, for a future extension.)

Roughly, signatures are type abstractions or interfaces of classes. Some other languages have similar facilities. C++ signatures are related to ML's signatures, Haskell's type classes, definition modules in Modula-2, interface modules in Modula-3, abstract types in Emerald, type modules in Trellis/Owl, categories in Scratchpad II, and types in POOL-I. For a more detailed discussion of signatures, see *Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++* by Gerald Baumgartner and Vincent F. Russo (Tech report CSD–TR–95–051, Dept. of Computer Sciences, Purdue University, August 1995, a slightly improved version appeared in *Software—Practice & Experience*, **25**(8), pp. 863–889, August 1995). You can get the tech report by anonymous FTP from `ftp.cs.purdue.edu` in '`pub/gb/Signature-design.ps.gz`'.

Syntactically, a signature declaration is a collection of member function declarations and nested type declarations. For example, this signature declaration defines a new abstract type `S` with member functions '`int foo ()`' and '`int bar (int)`':

```
signature S
{
  int foo ();
  int bar (int);
};
```

Since signature types do not include implementation definitions, you cannot write an instance of a signature directly. Instead, you can define a pointer to any class that contains the required interfaces as a *signature pointer*. Such a class *implements* the signature type.

To use a class as an implementation of `S`, you must ensure that the class has public member functions '`int foo ()`' and '`int bar (int)`'. The class can have other member functions as well, public or not; as long as it offers what's declared in the signature, it is suitable as an implementation of that signature type.

For example, suppose that `C` is a class that meets the requirements of signature `S` (`C` *conforms to* `S`). Then

```
C obj;
S * p = &obj;
```

defines a signature pointer `p` and initializes it to point to an object of type `C`. The member function call '`int i = p->foo ();`' executes '`obj.foo ()`'.

Abstract virtual classes provide somewhat similar facilities in standard C++. There are two main advantages to using signatures instead:

1. Subtyping becomes independent from inheritance. A class or signature type `T` is a subtype of a signature type `S` independent of any inheritance hierarchy as long as all the member functions declared in `S` are also found in `T`. So you can define a subtype hierarchy that is completely independent from any inheritance (implementation) hierarchy, instead of being forced to use types that mirror the class inheritance hierarchy.
2. Signatures allow you to work with existing class hierarchies as implementations of a signature type. If those class hierarchies are only available in compiled form, you're out of luck with abstract virtual classes, since an abstract virtual class cannot be retrofitted on top of existing class hierarchies. So you would be required to write interface classes as subtypes of the abstract virtual class.

There is one more detail about signatures. A signature declaration can contain member function *definitions* as well as member function declarations. A signature member function with a full definition is called a *default implementation*; classes need not contain that particular interface in order to conform. For example, a class `C` can conform to the signature

```
signature T
{
  int f (int);
  int f0 () { return f (0); };
};
```

whether or not `C` implements the member function '`int f0 ()`'. If you define `C::f0`, that definition takes precedence; otherwise, the default implementation `S::f0` applies.

6 gcov: a Test Coverage Program

gcov is a tool you can use in conjunction with GNU CC to test code coverage in your programs.

This chapter describes version 1.5 of gcov.

6.1 Introduction to gcov

gcov is a test coverage program. Use it in concert with GNU CC to analyze your programs to help create more efficient, faster running code. You can use gcov as a profiling tool to help discover where your optimization efforts will best affect your code. You can also use gcov along with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. gcov helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use gcov because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for 'hot spots' where the code is using a great deal of computer time. Likewise, because gcov accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful—they only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

gcov creates a logfile called '*sourcefile*.gcov' which indicates how many times each line of a source file '*sourcefile*.c' has executed. You can use these logfiles along with gprof to aid in fine-tuning the performance of your programs. gprof gives timing information you can use along with the information you get from gcov.

gcov works only on code compiled with GNU CC. It is not compatible with any other profiling or test coverage mechanism.

6.2 Invoking gcov

```
gcov [-b] [-v] [-n] [-l] [-f] [-o directory] sourcefile
```

`-b` 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.

`-v` Display the `gcov` version number (on the standard error stream).

`-n` Do not create the `gcov` output file.

`-l` Create long file names for included source files. For example, if the header file '`x.h`' contains code, and was included in the file '`a.c`', then running `gcov` on the file '`a.c`' will produce an output file called '`a.c.x.h.gcov`' instead of '`x.h.gcov`'. This can be useful if '`x.h`' is included in multiple source files.

`-f` Output summaries for each function in addition to the file level summary.

`-o` The directory where the object files live. Gcov will search for `.bb`, `.bbg`, and `.da` files in this directory.

When using `gcov`, you must first compile your program with two special GNU CC options: '`-fprofile-arcs -ftest-coverage`'. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by gcov. These additional files are placed in the directory where the source code is located.

Running the program will cause profile output to be generated. For each source file compiled with -fprofile-arcs, an accompanying `.da` file will be placed in the source directory.

Running `gcov` with your program's source file names as arguments will now produce a listing of the code along with frequency of execution for each line. For example, if your program is called '`tmp.c`', this is what you see when you use the basic `gcov` facility:

```
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
$ gcov tmp.c
 87.50% of 8 source lines executed in file tmp.c
Creating tmp.c.gcov.
```

The file '`tmp.c.gcov`' contains output from `gcov`. Here is a sample:

```
                main()
                {
           1      int i, total;

           1      total = 0;

          11      for (i = 0; i < 10; i++)
          10        total += i;

           1      if (total != 45)
      ######        printf ("Failure\n");
                  else
           1        printf ("Success\n");
```

```
                1    }
```

When you use the '`-b`' option, your output looks like this:

```
$ gcov -b tmp.c
 87.50% of 8 source lines executed in file tmp.c
 80.00% of 5 branches executed in file tmp.c
 80.00% of 5 branches taken at least once in file tmp.c
 50.00% of 2 calls executed in file tmp.c
Creating tmp.c.gcov.
```

Here is a sample of a resulting '`tmp.c.gcov`' file:

```
                     main()
                     {
           1           int i, total;

           1           total = 0;

          11           for (i = 0; i < 10; i++)
branch 0 taken = 91%
branch 1 taken = 100%
branch 2 taken = 100%
          10             total += i;

           1           if (total != 45)
branch 0 taken = 100%
      ######             printf ("Failure\n");
call 0 never executed
branch 1 never executed
                       else
           1             printf ("Success\n");
call 0 returns = 100%
           1         }
```

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions call `exit` or `longjmp`, and thus may not return everytime they are called.

The execution counts are cumulative. If the example program were executed again without removing the `.da` file, the count for the number of times each line in the source was executed would be added to the results of the previous run(s). This is potentially useful in several ways. For example, it could be used to accumulate data over a number of program runs as part of a test verification suite, or to provide more accurate long-term information over a large number of program runs.

The data in the `.da` files is saved immediately before the program exits. For each source file compiled with -fprofile-arcs, the profiling code first attempts to read in an existing `.da` file; if the file doesn't match the executable (differing number of basic block counts) it will ignore the contents of the file. It then adds in the new execution counts and finally writes the data to the file.

6.3 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 GNU CC options: '`-fprofile-arcs -ftest-coverage`'. Aside from that, you can use any other GNU CC options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for `gcov` to calculate separate execution counts for each line because there isn't separate code for each line. Hence the `gcov` output looks like this if you compiled the program with optimization:

```
      100  if (a != b)
      100    c = 1;
      100  else
      100    c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

6.4 Brief description of `gcov` data files

`gcov` uses three files for doing profiling. The names of these files are derived from the original *source* file by substituting the file suffix with either `.bb`, `.bbg`, or `.da`. All of these files are placed in the same directory as the source file, and contain data stored in a platform-independent method.

The `.bb` and `.bbg` files are generated when the source file is compiled with the GNU CC '`-ftest-coverage`' option. The `.bb` file contains a list of source files (including headers), functions within those files, and line numbers corresponding to each basic block in the source file.

The `.bb` file format consists of several lists of 4-byte integers which correspond to the line numbers of each basic block in the file. Each list is terminated by a line number of

0. A line number of -1 is used to designate that the source file name (padded to a 4-byte boundary and followed by another -1) follows. In addition, a line number of -2 is used to designate that the name of a function (also padded to a 4-byte boundary and followed by a -2) follows.

The `.bbg` file is used to reconstruct the program flow graph for the source file. It contains a list of the program flow arcs (possible branches taken from one basic block to another) for each function which, in combination with the `.bb` file, enables gcov to reconstruct the program flow.

In the `.bbg` file, the format is:

```
number of basic blocks for function #0 (4-byte number)
total number of arcs for function #0 (4-byte number)
count of arcs in basic block #0 (4-byte number)
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
destination basic block of arc #1 (4-byte number)
flag bits (4-byte number)
...
destination basic block of arc #N (4-byte number)
flag bits (4-byte number)
count of arcs in basic block #1 (4-byte number)
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
...
```

A -1 (stored as a 4-byte number) is used to separate each function's list of basic blocks, and to verify that the file has been read correctly.

The `.da` file is generated when a program containing object files built with the GNU CC '`-fprofile-arcs`' option is executed. A separate `.da` file is created for each source file compiled with this option, and the name of the `.da` file is stored as an absolute pathname in the resulting object file. This path name is derived from the source file name by substituting a `.da` suffix.

The format of the `.da` file is fairly simple. The first 8-byte number is the number of counts in the file, followed by the counts (stored as 8-byte numbers). Each count corresponds to the number of times each arc in the program is executed. The counts are cumulative; each time the program is executed, it attemps to combine the existing `.da` files with the new counts for this invocation of the program. It ignores the contents of any `.da` files whose number of arcs doesn't correspond to the current program, and merely overwrites them instead.

All three of these files use the functions in `gcov-io.h` to store integers; the functions in this header provide a machine-independent mechanism for storing and retrieving data from a stream.

7 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.

7.1 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.
- There are several obscure case of mis-using struct, union, and enum tags that are not detected as errors by the compiler.
- 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.
- Loop unrolling doesn't work properly for certain C++ programs. This is a bug in the C++ front end. It sometimes emits incorrect debug info, and the loop unrolling code is unable to recover from this error.

7.2 Installation Problems

This is a list of problems (and some apparent problems which don't really mean anything is wrong) that show up during installation of GNU CC.

- On certain systems, defining certain environment variables such as `CC` can interfere with the functioning of `make`.
- If you encounter seemingly strange errors when trying to build the compiler in a directory other than the source directory, it could be because you have previously configured the compiler in the source directory. Make sure you have done all the necessary preparations. See Section 3.3 [Other Dir], page 126.
- If you build GCC on a BSD system using a directory stored in a System V file system, problems may occur in running `fixincludes` if the System V file system doesn't support symbolic links. These problems result in a failure to fix the declaration of `size_t` in '`sys/types.h`'. If you find that `size_t` is a signed type and that type mismatches occur, this could be the cause.

  The solution is not to use such a directory for building GCC.
- In previous versions of GCC, the `gcc` driver program looked for `as` and `ld` in various places; for example, in files beginning with '`/usr/local/lib/gcc-`'. GCC version 2 looks for them in the directory '`/usr/local/lib/gcc-lib/`*target*`/`*version*'.

Thus, to use a version of `as` or `ld` that is not the system default, for example `gas` or GNU `ld`, you must put them in that directory (or make links to them from that directory).

- Some commands executed when making the compiler may fail (return a non-zero status) and be ignored by `make`. These failures, which are often due to files that were not found, are expected, and can safely be ignored.
- It is normal to have warnings in compiling certain files about unreachable code and about enumeration type clashes. These files' names begin with '`insn-`'. Also, '`real.c`' may get some warnings that you can ignore.
- Sometimes `make` recompiles parts of the compiler when installing the compiler. In one case, this was traced down to a bug in `make`. Either ignore the problem or switch to GNU Make.
- If you have installed a program known as purify, you may find that it causes errors while linking `enquire`, which is part of building GCC. The fix is to get rid of the file `real-ld` which purify installs—so that GCC won't try to use it.
- On GNU/Linux SLS 1.01, there is a problem with '`libc.a`': it does not contain the obstack functions. However, GCC assumes that the obstack functions are in '`libc.a`' when it is the GNU C library. To work around this problem, change the `__GNU_LIBRARY__` conditional around line 31 to '`#if 1`'.
- On some 386 systems, building the compiler never finishes because `enquire` hangs due to a hardware problem in the motherboard—it reports floating point exceptions to the kernel incorrectly. You can install GCC except for '`float.h`' by patching out the command to run `enquire`. You may also be able to fix the problem for real by getting a replacement motherboard. This problem was observed in Revision E of the Micronics motherboard, and is fixed in Revision F. It has also been observed in the MYLEX MXA-33 motherboard.

  If you encounter this problem, you may also want to consider removing the FPU from the socket during the compilation. Alternatively, if you are running SCO Unix, you can reboot and force the FPU to be ignored. To do this, type '`hd(40)unix auto ignorefpu`'.
- On some 386 systems, GCC crashes trying to compile '`enquire.c`'. This happens on machines that don't have a 387 FPU chip. On 386 machines, the system kernel is supposed to emulate the 387 when you don't have one. The crash is due to a bug in the emulator.

  One of these systems is the Unix from Interactive Systems: 386/ix. On this system, an alternate emulator is provided, and it does work. To use it, execute this command as super-user:

  ```
  ln /etc/emulator.rel1 /etc/emulator
  ```

  and then reboot the system. (The default emulator file remains present under the name '`emulator.dflt`'.)

  Try using '`/etc/emulator.att`', if you have such a problem on the SCO system.

  Another system which has this problem is Esix. We don't know whether it has an alternate emulator that works.

  On NetBSD 0.8, a similar problem manifests itself as these error messages:

```
enquire.c: In function ‘fprop’:
enquire.c:2328: floating overflow
```

- On SCO systems, when compiling GCC with the system's compiler, do not use '`-O`'. Some versions of the system's compiler miscompile GCC with '`-O`'.
- Sometimes on a Sun 4 you may observe a crash in the program `genflags` or `genoutput` while building GCC. This is said to be due to a bug in `sh`. You can probably get around it by running `genflags` or `genoutput` manually and then retrying the `make`.
- On Solaris 2, executables of GCC version 2.0.2 are commonly available, but they have a bug that shows up when compiling current versions of GCC: undefined symbol errors occur during assembly if you use '`-g`'.

  The solution is to compile the current version of GCC without '`-g`'. That makes a working compiler which you can use to recompile with '`-g`'.
- Solaris 2 comes with a number of optional OS packages. Some of these packages are needed to use GCC fully. If you did not install all optional packages when installing Solaris, you will need to verify that the packages that GCC needs are installed.

  To check whether an optional package is installed, use the `pkginfo` command. To add an optional package, use the `pkgadd` command. For further details, see the Solaris documentation.

  For Solaris 2.0 and 2.1, GCC needs six packages: '`SUNWarc`', '`SUNWbtool`', '`SUNWesu`', '`SUNWhea`', '`SUNWlibm`', and '`SUNWtoo`'.

  For Solaris 2.2, GCC needs an additional seventh package: '`SUNWsprot`'.
- On Solaris 2, trying to use the linker and other tools in '`/usr/ucb`' to install GCC has been observed to cause trouble. For example, the linker may hang indefinitely. The fix is to remove '`/usr/ucb`' from your `PATH`.
- If you use the 1.31 version of the MIPS assembler (such as was shipped with Ultrix 3.1), you will need to use the -fno-delayed-branch switch when optimizing floating point code. Otherwise, the assembler will complain when the GCC compiler fills a branch delay slot with a floating point instruction, such as `add.d`.
- If on a MIPS system you get an error message saying "does not have gp sections for all it's [sic] sectons [sic]", don't worry about it. This happens whenever you use GAS with the MIPS linker, but there is not really anything wrong, and it is okay to use the output file. You can stop such warnings by installing the GNU linker.

  It would be nice to extend GAS to produce the gp tables, but they are optional, and there should not be a warning about their absence.
- In Ultrix 4.0 on the MIPS machine, '`stdio.h`' does not work with GNU CC at all unless it has been fixed with `fixincludes`. This causes problems in building GCC. Once GCC is installed, the problems go away.

  To work around this problem, when making the stage 1 compiler, specify this option to Make:

  ```
  GCC_FOR_TARGET="./xgcc -B./ -I./include"
  ```

  When making stage 2 and stage 3, specify this option:

  ```
  CFLAGS="-g -I./include"
  ```

- Users have reported some problems with version 2.0 of the MIPS compiler tools that were shipped with Ultrix 4.1. Version 2.10 which came with Ultrix 4.2 seems to work fine.

  Users have also reported some problems with version 2.20 of the MIPS compiler tools that were shipped with RISC/os 4.x. The earlier version 2.11 seems to work fine.
- Some versions of the MIPS linker will issue an assertion failure when linking code that uses `alloca` against shared libraries on RISC-OS 5.0, and DEC's OSF/1 systems. This is a bug in the linker, that is supposed to be fixed in future revisions. To protect against this, GCC passes '`-non_shared`' to the linker unless you pass an explicit '`-shared`' or '`-call_shared`' switch.
- On System V release 3, you may get this error message while linking:

  ```
  ld fatal: failed to write symbol name something
   in strings table for file whatever
  ```

  This probably indicates that the disk is full or your ULIMIT won't allow the file to be as large as it needs to be.

  This problem can also result because the kernel parameter `MAXUMEM` is too small. If so, you must regenerate the kernel and make the value much larger. The default value is reported to be 1024; a value of 32768 is said to work. Smaller values may also work.
- On System V, if you get an error like this,

  ```
  /usr/local/lib/bison.simple: In function 'yyparse':
  /usr/local/lib/bison.simple:625: virtual memory exhausted
  ```

  that too indicates a problem with disk space, ULIMIT, or `MAXUMEM`.
- Current GCC versions probably do not work on version 2 of the NeXT operating system.
- On NeXTStep 3.0, the Objective C compiler does not work, due, apparently, to a kernel bug that it happens to trigger. This problem does not happen on 3.1.
- On the Tower models 4*n*0 and 6*n*0, by default a process is not allowed to have more than one megabyte of memory. GCC cannot compile itself (or many other programs) with '`-O`' in that much memory.

  To solve this problem, reconfigure the kernel adding the following line to the configuration file:

  ```
  MAXUMEM = 4096
  ```
- On HP 9000 series 300 or 400 running HP-UX release 8.0, there is a bug in the assembler that must be fixed before GCC can be built. This bug manifests itself during the first stage of compilation, while building '`libgcc2.a`':

  ```
  _floatdisf
  cc1: warning: '-g' option not supported on this version of GCC
  cc1: warning: '-g1' option not supported on this version of GCC
  ./xgcc: Internal compiler error: program as got fatal signal 11
  ```

  A patched version of the assembler is available by anonymous ftp from `altdorf.ai.mit.edu` as the file '`archive/cph/hpux-8.0-assembler`'. If you have HP software support, the patch can also be obtained directly from HP, as described in the following note:

  > This is the patched assembler, to patch SR#1653-010439, where the assembler aborts on floating point constants.

> The bug is not really in the assembler, but in the shared library version of the function “cvtnum(3c)”. The bug on “cvtnum(3c)” is SR#4701-078451. Anyway, the attached assembler uses the archive library version of “cvtnum(3c)” and thus does not exhibit the bug.

  This patch is also known as PHCO_4484.

- On HP-UX version 8.05, but not on 8.07 or more recent versions, the `fixproto` shell script triggers a bug in the system shell. If you encounter this problem, upgrade your operating system or use BASH (the GNU shell) to run `fixproto`.
- Some versions of the Pyramid C compiler are reported to be unable to compile GCC. You must use an older version of GCC for bootstrapping. One indication of this problem is if you get a crash when GCC compiles the function `muldi3` in file ‘`libgcc2.c`’.

  You may be able to succeed by getting GCC version 1, installing it, and using it to compile GCC version 2. The bug in the Pyramid C compiler does not seem to affect GCC version 1.
- There may be similar problems on System V Release 3.1 on 386 systems.
- On the Intel Paragon (an i860 machine), if you are using operating system version 1.0, you will get warnings or errors about redefinition of `va_arg` when you build GCC.

  If this happens, then you need to link most programs with the library ‘`iclib.a`’. You must also modify ‘`stdio.h`’ as follows: before the lines

  ```
  #if     defined(__i860__) && !defined(_VA_LIST)
  #include <va_list.h>
  ```

  insert the line

  ```
  #if __PGC__
  ```

  and after the lines

  ```
  extern int  vprintf(const char *, va_list );
  extern int  vsprintf(char *, const char *, va_list );
  #endif
  ```

  insert the line

  ```
  #endif /* __PGC__ */
  ```

  These problems don’t exist in operating system version 1.1.
- On the Altos 3068, programs compiled with GCC won’t work unless you fix a kernel bug. This happens using system versions V.2.2 1.0gT1 and V.2.2 1.0e and perhaps later versions as well. See the file ‘`README.ALTOS`’.
- You will get several sorts of compilation and linking errors on the we32k if you don’t follow the special instructions. See Section 3.2 [Configurations], page 111.
- A bug in the HP-UX 8.05 (and earlier) shell will cause the fixproto program to report an error of the form:

  ```
  ./fixproto: sh internal 1K buffer overflow
  ```

  To fix this, change the first line of the fixproto script to look like:

  ```
  #!/bin/ksh
  ```

7.3 Cross-Compiler Problems

You may run into problems with cross compilation on certain machines, for several reasons.

- Cross compilation can run into trouble for certain machines because some target machines' assemblers require floating point numbers to be written as *integer* constants in certain contexts.

  The compiler writes these integer constants by examining the floating point value as an integer and printing that integer, because this is simple to write and independent of the details of the floating point representation. But this does not work if the compiler is running on a different machine with an incompatible floating point format, or even a different byte-ordering.

  In addition, correct constant folding of floating point values requires representing them in the target machine's format. (The C standard does not quite require this, but in practice it is the only way to win.)

  It is now possible to overcome these problems by defining macros such as `REAL_VALUE_TYPE`. But doing so is a substantial amount of work for each target machine. See section "Cross Compilation and Floating Point Format" in *Using and Porting GCC*.
- At present, the program '`mips-tfile`' which adds debug support to object files on MIPS systems does not work in a cross compile environment.

7.4 Interoperation

This section lists various difficulties encountered in using GNU C or GNU C++ together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

- Objective C does not work on the RS/6000.
- GNU C++ does not do name mangling in the same way as other C++ compilers. This means that object files compiled with one compiler cannot be used with another.

  This effect is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation, including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers—but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.
- Older GDB versions sometimes fail to read the output of GCC version 2. If you have trouble, get GDB version 4.4 or later.
- DBX rejects some files produced by GCC, though it accepts similar constructs in output from PCC. Until someone can supply a coherent description of what is valid DBX input and what is not, there is nothing I can do about these problems. You are on your own.
- The GNU assembler (GAS) does not support PIC. To generate PIC code, you must use some other assembler, such as '`/bin/as`'.
- On some BSD systems, including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.

- Use of '`-I/usr/include`' may cause trouble.

  Many systems come with header files that won't work with GCC unless corrected by `fixincludes`. The corrected header files go in a new directory; GCC searches this directory before '`/usr/include`'. If you use '`-I/usr/include`', this tells GCC to search '`/usr/include`' earlier on, before the corrected headers. The result is that you get the uncorrected header files.

  Instead, you should use these options (when compiling C programs):

  ```
  -I/usr/local/lib/gcc-lib/target/version/include -I/usr/include
  ```

  For C++ programs, GCC also uses a special directory that defines C++ interfaces to standard C subroutines. This directory is meant to be searched *before* other standard include directories, so that it takes precedence. If you are compiling C++ programs and specifying include directories explicitly, use this option first, then the two options above:

  ```
  -I/usr/local/lib/g++-include
  ```

- 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 GNU CC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function `access_double` instead of directly with '`*`':

  ```
  inline double
  access_double (double *unaligned_ptr)
  {
    union d2i { double d; int i[2]; };

    union d2i *p = (union d2i *) unaligned_ptr;
    union d2i u;

    u.i[0] = p->i[0];
    u.i[1] = p->i[1];

    return u.d;
  }
  ```

  Storing into the pointer can be done likewise with the same union.

- On Solaris, the `malloc` function in the '`libmalloc.a`' library may allocate memory that is only 4 byte aligned. Since GCC on the Sparc assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the '`libmalloc.a`' library.

The solution is to not use the '`libmalloc.a`' library. Use instead `malloc` and related functions from '`libc.a`'; they do not have this problem.

- Sun forgot to include a static version of '`libdl.a`' with some versions of SunOS (mainly 4.1). This results in undefined symbols when linking static binaries (that is, if you use '`-static`'). If you see undefined symbols `_dlclose`, `_dlsym` or `_dlopen` when linking, compile and link against the file '`mit/util/misc/dlsym.c`' from the MIT version of X windows.
- The 128-bit long double format that the Sparc port supports currently works by using the architecturally defined quad-word floating point instructions. Since there is no hardware that supports these instructions they must be emulated by the operating system. Long doubles do not work in Sun OS versions 4.0.3 and earlier, because the kernel emulator uses an obsolete and incompatible format. Long doubles do not work in Sun OS version 4.1.1 due to a problem in a Sun library. Long doubles do work on Sun OS versions 4.1.2 and higher, but GCC does not enable them by default. Long doubles appear to work in Sun OS 5.x (Solaris 2.x).
- On HP-UX version 9.01 on the HP PA, the HP compiler `cc` does not compile GCC correctly. We do not yet know why. However, GCC compiled on earlier HP-UX versions works properly on HP-UX 9.01 and can compile itself properly on 9.01.
- On the HP PA machine, ADB sometimes fails to work on functions compiled with GCC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GCC doesn't generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.
- Debugging ('`-g`') is not supported on the HP PA machine, unless you use the preliminary GNU tools (see Chapter 3 [Installation], page 103).
- 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.
- The current version of the assembler ('`/bin/as`') for the RS/6000 has certain problems that prevent the '`-g`' option in GCC from working. Note that '`Makefile.in`' uses '`-g`' by default when compiling '`libgcc2.c`'.

IBM has produced a fixed version of the assembler. The upgraded assembler unfortunately was not included in any of the AIX 3.2 update PTF releases (3.2.2, 3.2.3, or 3.2.3e). Users of AIX 3.1 should request PTF U403044 from IBM and users of AIX 3.2 should request PTF U416277. See the file '`README.RS6000`' for more details on these updates.

You can test for the presense of a fixed assembler by using the command

```
as -u < /dev/null
```

If the command exits normally, the assembler fix already is installed. If the assembler complains that "-u" is an unknown flag, you need to order the fix.

- 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 ANSI C.

- 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 the RS/6000, XLC version 1.3.0.0 will miscompile '`jump.c`'. XLC version 1.3.0.1 or later fixes this problem. You can obtain XLC-1.3.0.2 by requesting PTF 421749 from IBM.
- There is an assembler bug in versions of DG/UX prior to 5.4.2.01 that occurs when the '`fldcr`' instruction is used. GCC uses '`fldcr`' on the 88100 to serialize volatile memory references. Use the option '`-mno-serialize-volatile`' if your version of the assembler has this bug.
- On VMS, GAS versions 1.38.1 and earlier may cause spurious warning messages from the linker. These warning messages complain of mismatched psect attributes. You can ignore them. See Section 3.6 [VMS Install], page 131.
- On NewsOS version 3, if you include both of the files '`stddef.h`' and '`sys/types.h`', you get an error because there are two typedefs of `size_t`. You should change '`sys/types.h`' by adding these lines around the definition of `size_t`:

```
#ifndef _SIZE_T
#define _SIZE_T
actual typedef here
#endif
```

- On the Alliant, the system's own convention for returning structures and unions is unusual, and is not compatible with GCC no matter what options are used.
- On the IBM RT PC, the MetaWare HighC compiler (hc) uses a different convention for structure and union returning. Use the option '`-mhc-struct-return`' to tell GCC to use a convention compatible with it.
- On Ultrix, the Fortran compiler expects registers 2 through 5 to be saved by function calls. However, the C compiler uses conventions compatible with BSD Unix: registers 2 through 5 may be clobbered by function calls.

  GCC uses the same convention as the Ultrix C compiler. You can use these options to produce code compatible with the Fortran compiler:

  ```
  -fcall-saved-r2 -fcall-saved-r3 -fcall-saved-r4 -fcall-saved-r5
  ```
- On the WE32k, you may find that programs compiled with GCC do not work with the standard shared C library. You may need to link with the ordinary C compiler. If you do so, you must specify the following options:

  ```
  -L/usr/local/lib/gcc-lib/we32k-att-sysv/2.8.1 -lgcc -lc_s
  ```

  The first specifies where to find the library '`libgcc.a`' specified with the '`-lgcc`' option.

  GCC does linking by invoking `ld`, just as `cc` does, and there is no reason why it *should* matter which compilation program you use to invoke `ld`. If someone tracks this problem down, it can probably be fixed easily.
- On the Alpha, you may get assembler errors about invalid syntax as a result of floating point constants. This is due to a bug in the C library functions `ecvt`, `fcvt` and `gcvt`. Given valid floating point numbers, they sometimes print '`NaN`'.
- On Irix 4.0.5F (and perhaps in some other versions), an assembler bug sometimes reorders instructions incorrectly when optimization is turned on. If you think this may be happening to you, try using the GNU assembler; GAS version 2.1 supports ECOFF on Irix.

  Or use the '`-noasmopt`' option when you compile GCC with itself, and then again when you compile your program. (This is a temporary kludge to turn off assembler optimization on Irix.) If this proves to be what you need, edit the assembler spec in the file '`specs`' so that it unconditionally passes '`-O0`' to the assembler, and never passes '`-O2`' or '`-O3`'.

7.5 Problems Compiling Certain Programs

Certain programs have problems compiling.

- Parse errors may occur compiling X11 on a Decstation running Ultrix 4.2 because of problems in DEC's versions of the X11 header files '`X11/Xlib.h`' and '`X11/Xutil.h`'. People recommend adding '`-I/usr/include/mit`' to use the MIT versions of the header files, using the '`-traditional`' switch to turn off ANSI C, or fixing the header files by adding this:

  ```
  #ifdef __STDC__
  #define NeedFunctionPrototypes 0
  #endif
  ```
- If you have trouble compiling Perl on a SunOS 4 system, it may be because Perl specifies '`-I/usr/ucbinclude`'. This accesses the unfixed header files. Perl specifies the options

```
-traditional -Dvolatile=__volatile__
-I/usr/include/sun -I/usr/ucbinclude
-fpcc-struct-return
```

most of which are unnecessary with GCC 2.4.5 and newer versions. You can make a properly working Perl by setting `ccflags` to '`-fwritable-strings`' (implied by the '`-traditional`' in the original options) and `cppflags` to empty in '`config.sh`', then typing '`./doSH; make depend; make`'.

- 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
  ```

7.6 Incompatibilities of GCC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The '`-traditional`' option eliminates many of these incompatibilities, *but not all*, by telling GNU C to behave like the other C compilers.

- GCC normally makes string constants read-only. If several identical-looking string constants are used, GCC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '`-fwritable-strings`' flag, which directs GCC to handle string constants the same way most C compilers do. '`-traditional`' also has this effect, among others.

- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ANSI C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.

- GCC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GCC

  ```
  #define foo(a) "a"
  ```

will produce output `"a"` regardless of what the argument *a* is.

The '`-traditional`' option directs GCC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {
    int a, b;

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

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

  Here `a` may or may not be restored to its first value when the `longjmp` occurs. If `a` is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

  If you use the '`-W`' option with the '`-O`' option, you will get a warning when GCC thinks such a problem might be possible.

  The '`-traditional`' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GCC. For example, a program like this will not work:

  ```
  foobar (
  #define luser
          hack)
  ```

  ANSI C does not permit such a construct. It would make sense to support it when '`-traditional`' is used, but it is too much work to implement.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a `extern` declaration affects all the rest of the file even if it happens within a block.

  The '`-traditional`' option directs GNU C to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.
- PCC allows whitespace in the middle of compound assignment operators such as '`+=`'. GCC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.
- GCC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GCC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '`/*...*/`'. However, '`-traditional`' suppresses these error messages.
- Many user programs contain the declaration '`long time ();`'. In the past, the system header files on many systems did not actually declare `time`, so it did not matter what type your program declared it to return. But in systems with ANSI C headers, `time` is declared to return `time_t`, and if that is not the same as `long`, then '`long time ();`' is erroneous.

  The solution is to change your program to use `time_t` as the return type of `time`.
- When compiling functions that return `float`, PCC converts it to a double. GCC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.
- When compiling functions that return structures or unions, GCC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GCC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GCC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GCC where to pass this address.

  By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GCC does not use this method because it is slower and nonreentrant.

  On some newer machines, PCC uses a reentrant convention for all structure and union returning. GCC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

You can tell GCC to use a compatible convention for all structure and union returning with the option '`-fpcc-struct-return`'.

- GNU C complains about program fragments such as '`0x74ae-0x4000`' which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single *preprocessing token*. Each such token must correspond to one token in C. Since this does not, GNU C prints an error message. Although it may appear obvious that what is meant is an operator and two values, the ANSI C standard specifically requires that this be treated as erroneous.

  A *preprocessing token* is a *preprocessing number* if it begins with a digit and is followed by letters, underscores, digits, periods and '`e+`', '`e-`', '`E+`', or '`E-`' character sequences.

  To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

7.7 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 ANSI 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.

7.8 Standard Libraries

GCC by itself attempts to be what the ISO/ANSI C standard calls a *conforming freestanding implementation*. This means all ANSI C language features are available, as well as the contents of '`float.h`', '`limits.h`', '`stdarg.h`', and '`stddef.h`'. 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`) has been ported to a number of operating systems, and provides ANSI/ISO, POSIX, BSD and SystemV compatibility. You could also ask your operating system vendor if newer libraries are available.

7.9 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 ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

- Accesses to bitfields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bitfield; it may even vary for a given bitfield according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bitfields.
- GCC comes with shell scripts to fix certain known problems in system header files. They install corrected copies of various header files in a special directory where only GCC will normally look for them. The scripts adapt to various systems by searching all the system header files for the problem cases that we know about.

  If new system header files are installed, nothing automatically arranges to update the corrected header files. You will have to reinstall GCC to fix the new header files. More specifically, go to the build directory and delete the files '`stmp-fixinc`' and '`stmp-headers`', and the subdirectory `include`; then do '`make install`' again.
- On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the '`-ffloat-store`' option (see Section 2.8 [Optimize Options], page 35).
- On the MIPS, variable argument functions using '`varargs.h`' cannot have a floating point value for the first argument. The reason for this is that in the absence of a prototype in scope, if the first argument is a floating point, it is passed in a floating point register, rather than an integer register.

  If the code is rewritten to use the ANSI standard '`stdarg.h`' method of variable arguments, and the prototype is in scope at the time of the call, everything will work fine.
- On the H8/300 and H8/300H, variable argument functions must be implemented using the ANSI standard '`stdarg.h`' method of variable arguments. Furthermore, calls to functions using '`stdarg.h`' variable arguments must have a prototype for the called function in scope at the time of the call.

7.10 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.

7.10.1 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 draft ANSI standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to g++ from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: g++ reports as undefined symbols any static data members that lack definitions.

7.10.2 Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like string classes, especially ones that define a conversion function to type char * or const char * – which is one reason why the standard string class requires you to call the c_str member function. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function strfunc that returns string objects, and another function charfunc that operates on pointers to char:

```
string strfunc ();
void charfunc (const char *);

void
f ()
{
  const char *p = strfunc().c_str();
  ...
  charfunc (p);
  ...
  charfunc (p);
}
```

In this situation, it may seem reasonable to save a pointer to the C string returned by the c_str member function and use that rather than call c_str repeatedly. However, the temporary string created by the call to strfunc is destroyed after p is initialized, at which point p is left pointing to freed memory.

Code like this may run successfully under some other compilers, particularly obsolete cfront-based compilers that delete temporaries along with normal local variables. However, the GNU C++ behavior is standard-conforming, so if your program depends on late destruction of temporaries it is not portable.

The safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
string& tmp = strfunc ();
charfunc (tmp.c_str ());
```

7.10.3 Implicit Copy-Assignment for Virtual Bases

When a base class is virtual, only one subobject of the base class belongs to each full object. Also, the constructors and destructors are invoked only once, and called from the most-derived class. However, such objects behave unspecified when being assigned. For example:

```
struct Base{
  char *name;
  Base(char *n) : name(strdup(n)){}
  Base& operator= (const Base& other){
   free (name);
   name = strdup (other.name);
  }
};

struct A:virtual Base{
  int val;
  A():Base("A"){}
};

struct B:virtual Base{
  int bval;
  B():Base("B"){}
};

struct Derived:public A, public B{
  Derived():Base("Derived"){}
};

void func(Derived &d1, Derived &d2)
{
  d1 = d2;
}
```

The C++ standard specifies that '`Base::Base`' is only called once when constructing or copy-constructing a Derived object. It is unspecified whether '`Base::operator=`' is called more than once when the implicit copy-assignment for Derived objects is invoked (as it is inside '`func`' in the example).

g++ implements the "intuitive" algorithm for copy-assignment: assign all direct bases, then assign all members. In that algorithm, the virtual base subobject can be encountered many times. In the example, copying proceeds in the following order: '`val`', '`name`' (via `strdup`), '`bval`', and '`name`' again.

If application code relies on copy-assignment, a user-defined copy-assignment operator removes any uncertainties. With such an operator, the application can define whether and how the virtual base subobject is assigned.

7.11 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. ANSI 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 recommand 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.)

7.12 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 about unreachable code.

  It's very common to have unreachable code in machine-generated programs. For example, this happens normally in some files of GNU C itself.
- Warning when a non-void function value is ignored.

  Coming as I do from a Lisp background, I balk at the idea that there is something dangerous about discarding a value. There are functions that return values which some callers may find useful; it makes no sense to clutter the program with a cast to `void` whenever the value isn't useful.
- Assuming (for optimization) that the address of an external symbol is never zero.

  This assumption is false on certain systems when '`#pragma weak`' is used.
- 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 bitfields unsigned by default on particular machines where "the ABI standard" says to do so.

The ANSI C standard leaves it up to the implementation whether a bitfield 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 bitfields 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 bitfields as well.

Some computer manufacturers have published Application Binary Interface standards which specify that plain bitfields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bitfields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bitfields or unsigned is of no concern to other object files, even if they access the same bitfields 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 bitfields 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 bitfields in the same fashion on all types of machines (by default).

There are some arguments for making bitfields 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 bitfield whether it is signed or not. In this way, they write programs which have the same meaning in both C dialects.)

- Undefining `__STDC__` when '`-ansi`' is not used.

  Currently, GCC defines `__STDC__` as long as you don't use '`-traditional`'. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ANSI C, such as function prototypes or ANSI token concatenation. Since plain '`gcc`' supports all the features of ANSI 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 ANSI C program, because the ANSI 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 ANSI C library.

Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ANSI C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that claim to support ANSI C, such as '`gcc -ansi`'—not for other compilers such as plain '`gcc`'. Whatever the ANSI 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. 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 ANSI 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 ANSI 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 ANSI 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.

7.13 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-ANSI constructs get warnings or errors.

See Section 2.6 [Options to Request or Suppress Warnings], page 22, for more detail on these and related command-line options.

8 Reporting Bugs

Your bug reports play an essential role in making GCC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 7 [Trouble], page 193. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. (If it does not, look in the service directory; see Chapter 9 [Service], page 225.) In any case, the principal function of a bug report is to help the entire community by making the next version of GCC work better. Bug reports are your contribution to the maintenance of GCC.

Since the maintainers are very overloaded, we cannot respond to every bug report. However, if the bug has not been fixed, we are likely to send you a patch and ask you to tell us whether it works.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

8.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C and traditional C (see Section 7.6 [Incompatibilities], page 203). These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C or C++ compiler.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results. But the value of the function is undefined if `return` is omitted; it is not a bug when GCC produces different results.

  Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GCC might interpret it another way. Neither compiler is wrong. The bug is in your code.

  After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.
- If the compiler produces an error message for valid input, that is a compiler bug.

- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of C or C++ (or Fortran or Objective-C) compilers, your suggestions for improvement of GCC are welcome in any case.

8.2 Where to Report Bugs

Send bug reports for the GNU Compiler Collection to '`gcc-bugs@gcc.gnu.org`'. In accordance with the GNU-wide convention, in which bug reports for tool "foo" are sent to '`bug-foo@gnu.org`', the address '`bug-gcc@gnu.org`' may also be used; it will forward to the address given above.

Please read '`<URL:http://www.gnu.org/software/gcc/bugs.html>`' for bug reporting instructions before you post a bug report.

Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: a newsgroup posting does not contain a mail path back to the sender. Thus, if maintainers need more information, they may be unable to reach you. For this reason, you should always send bug reports by mail to the proper mailing list.

As a last resort, send bug reports on paper to:

```
GNU Compiler Bugs
Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA
```

8.3 How to Report Bugs

You may find additional and/or more up-to-date instructions at '`<URL:http://www.gnu.org/software/gcc`

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

To enable someone to investigate the bug, you should include all these things:

- The version of GCC. You can get this by running it with the '`-v`' option.

  Without this, we won't know whether there is any point in looking for the bug in the current version of GCC.

- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send a source file and any header files that it requires. If the bug is in the compiler proper ('`cc1`'), send the preprocessor output generated by adding '`-save-temps`' to the compilation command (see Section 2.7 [Debugging Options], page 29). When you do this, use the same '`-I`', '`-D`' or '`-U`' options that you used in actual compilation. Then send the *input*.i or *input*.ii files generated.

  A single statement is not enough of an example. In order to compile it, it must be embedded in a complete file of compiler input; and the bug might depend on the details of how this is done.

  Without a real example one can compile, all anyone can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.

  Even if the input file that fails comes from a GNU program, you should still send the complete test case. Don't ask the GCC maintainers to do the extra work of obtaining the program in question—they are all overworked as it is. Also, the problem may depend on what is in the header files on your system; it is unreliable for the GCC maintainers to try the problem with the header files available to them. By sending CPP output, you can eliminate this source of uncertainty and save us a certain percentage of wild goose chases.

- The command arguments you gave GCC to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list all the options.

  If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.

- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed the compiler.
- A complete list of any modifications you have made to the compiler source. (We don't promise to investigate the bug unless it happens in an unmodified compiler. But if you've made modifications and don't tell us, then you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own (such as a machine description for a machine we don't support) is a modification of the compiler source.

- Details of any other deviations from the standard procedure for installing GCC.
- A description of what behavior you observe that you believe is incorrect. For example, "The compiler gets a fatal signal," or, "The assembler instruction at line 208 in the output is incorrect."

  Of course, if the bug is that the compiler gets a fatal signal, then one can't miss it. But if the bug is incorrect output, the maintainer might not notice unless it is glaringly wrong. None of us has time to study all the assembler code from a 50-line C program just on the chance that one instruction might be wrong. We need *you* to do this part!

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here would not. If you *said* to expect a crash, then when the compiler here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

  If the problem is a diagnostic when compiling GCC with some other compiler, say whether it is a warning or an error.

  Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information unless the program is short and simple. None of us has time to study a large program to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell us which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as finding a bug in the program itself.
- If you send examples of assembler code output from GCC, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to mention something in the GCC source, refer to it by context, not by line number.

  The line numbers in the development sources don't match those in your sources. Your line numbers would convey no useful information to the maintainers.
- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to have any chance of being useful.

  For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GCC because the compiler is largely data-driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

  Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

  In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop—which insn it has reached—is usually in a local variable, not in an argument.

What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GCC bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions may affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.

- In particular, some people insert conditionals '`#ifdef BUG`' around a statement which, if removed, makes the bug not happen. These are just clutter; we won't pay any attention to them anyway. Besides, you should send us cpp output, and that can't have conditionals.

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as GCC it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send the example, we won't be able to construct one, so we won't be able to verify that the bug is fixed.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we won't install it. A test case will help us to understand.

  See Section 8.4 [Sending Patches], page 222, for guidelines on how to make it easy for us to understand and install your patches.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without first using the debugger to find the facts.

- A core dump file.

  We have no way of examining a core dump for your type of machine unless we have an identical system—and if we do have one, we should be able to reproduce the crash ourselves.

8.4 Sending Patches for GCC

If you would like to write bug fixes or improvements for the GNU C compiler, that is very helpful. Send suggested fixes to the patches mailing list, `gcc-patches@gcc.gnu.org`.

Please follow these guidelines so we can study your patches efficiently. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU C is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

  (Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them—to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then the two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.

  Ideally, each change you send should be impossible to subdivide into parts that we might want to consider separately, because each of its parts gets its motivation from the other parts.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.
- Use '`diff -c`' to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unidiff format is better than contextless diffs, but not as easy to read as '`-c`' format.

If you have GNU diff, use '`diff -cp`', which shows the name of the function that each change occurs in.

- Write the change log entries for your changes. We get lots of changes, and we don't have time to do all the change log writing ourselves.

  Read the '`ChangeLog`' file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

  If you would like your name to appear in the header line for who made the change, send us the header line.

- When you write the fix, keep in mind that we can't install a change that would break other systems.

  People often suggest fixing a problem by changing machine-independent files such as '`toplev.c`' to do something special that a particular system needs. Sometimes it is totally obvious that such changes would break GCC for almost all users. We can't possibly make a change like that. At best it might tell us how to write another patch that would solve the problem acceptably.

  Sometimes people send fixes that *might* be an improvement in general—but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is good to install.

9 How To Get Help with GCC

If you need help installing, using or changing GCC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `gcc-bugs@gcc.gnu.org` or `bug-gcc@gnu.org`, and if that brings no response, try `gcc@gcc.gnu.org`.
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named '`SERVICE`' in the GCC distribution.

10 Contributing to GCC Development

If you would like to help pretest GCC releases to assure they work well, or if you would like to work on improving GCC, please contact the maintainers at `gcc@gcc.gnu.org`. A pretester should be willing to try to investigate bugs as well as report them.

If you'd like to work on improvements, please ask for suggested projects or suggest your own ideas. If you have already written an improvement, please tell us about it. If you have not yet started work, it is useful to contact `gcc@gcc.gnu.org` before you start; the maintainers may be able to suggest ways to make your extension fit in better with the rest of GCC and with other development plans.

11 Using GCC on VMS

Here is how to use GCC on VMS.

11.1 Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GCC attempts to translate file names in '`#include`' into names that VMS will understand. The basic strategy is to prepend a prefix to the specification of the include file, convert the whole filename to a VMS filename, and then try to open the file. GCC tries various prefixes one by one until one of them succeeds:

1. The first prefix is the '`GNU_CC_INCLUDE:`' logical name: this is where GNU C header files are traditionally stored. If you wish to store header files in non-standard locations, then you can assign the logical '`GNU_CC_INCLUDE`' to be a search list, where each element of the list is suitable for use with a rooted logical.
2. The next prefix tried is '`SYS$SYSROOT:[SYSLIB.]`'. This is where VAX-C header files are traditionally stored.
3. If the include file specification by itself is a valid VMS filename, the preprocessor then uses this name with no prefix in an attempt to open the include file.
4. If the file specification is not a valid VMS filename (i.e. does not contain a device or a directory specifier, and contains a '`/`' character), the preprocessor tries to convert it from Unix syntax to VMS syntax.

   Conversion works like this: the first directory name becomes a device, and the rest of the directories are converted into VMS-format directory names. For example, the name '`X11/foobar.h`' is translated to '`X11:[000000]foobar.h`' or '`X11:foobar.h`', whichever one can be opened. This strategy allows you to assign a logical name to point to the actual location of the header files.
5. If none of these strategies succeeds, the '`#include`' fails.

Include directives of the form:

```
#include foobar
```

are a common source of incompatibility between VAX-C and GCC. VAX-C treats this much like a standard `#include <foobar.h>` directive. That is incompatible with the ANSI C behavior implemented by GCC: to expand the name `foobar` as a macro. Macro expansion should eventually yield one of the two standard formats for `#include`:

```
#include "file"
#include <file>
```

If you have this problem, the best solution is to modify the source to convert the `#include` directives to one of the two standard forms. That will work with either compiler. If you want a quick and dirty fix, define the file names as macros with the proper expansion, like this:

```
#define stdio <stdio.h>
```

This will work, as long as the name doesn't conflict with anything else in the program.

Another source of incompatibility is that VAX-C assumes that:

```
#include "foobar"
```

is actually asking for the file 'foobar.h'. GCC does not make this assumption, and instead takes what you ask for literally; it tries to read the file 'foobar'. The best way to avoid this problem is to always specify the desired file extension in your include directives.

GCC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GCC distribution does not contain header files to define constants and structures for some VMS system-specific functions, there is no reason why you cannot use GCC with any of these functions. You first may have to generate or create header files, either by using the public domain utility `UNSDL` (which can be found on a DECUS tape), or by extracting the relevant modules from one of the system macro libraries, and using an editor to construct a C header file.

A `#include` file name cannot contain a DECNET node name. The preprocessor reports an I/O error if you attempt to use a node name, whether explicitly, or implicitly via a logical name.

11.2 Global Declarations and VMS

GCC does not provide the `globalref`, `globaldef` and `globalvalue` keywords of VAX-C. You can get the same effect with an obscure feature of GAS, the GNU assembler. (This requires GAS version 1.39 or later.) The following macros allow you to use this feature in a fairly natural way:

```
#ifdef __GNUC__
#define GLOBALREF(TYPE,NAME)                       \
  TYPE NAME                                        \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME)
#define GLOBALDEF(TYPE,NAME,VALUE)                 \
  TYPE NAME                                        \
  asm ("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME) \
    = VALUE
#define GLOBALVALUEREF(TYPE,NAME)                  \
  const TYPE NAME[1]                               \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)
#define GLOBALVALUEDEF(TYPE,NAME,VALUE)            \
  const TYPE NAME[1]                               \
  asm ("_$$PsectAttributes_GLOBALVALUE$$" #NAME)  \
    = {VALUE}
#else
#define GLOBALREF(TYPE,NAME) \
  globalref TYPE NAME
#define GLOBALDEF(TYPE,NAME,VALUE) \
  globaldef TYPE NAME = VALUE
#define GLOBALVALUEDEF(TYPE,NAME,VALUE) \
  globalvalue TYPE NAME = VALUE
#define GLOBALVALUEREF(TYPE,NAME) \
  globalvalue TYPE NAME
#endif
```

(The _$$PsectAttributes_GLOBALSYMBOL prefix at the start of the name is removed by the assembler, after it has modified the attributes of the symbol). These macros are provided in the VMS binaries distribution in a header file 'GNU_HACKS.H'. An example of the usage is:

```
GLOBALREF (int, ijk);
GLOBALDEF (int, jkl, 0);
```

The macros GLOBALREF and GLOBALDEF cannot be used straightforwardly for arrays, since there is no way to insert the array dimension into the declaration at the right place. However, you can declare an array with these macros if you first define a typedef for the array type, like this:

```
typedef int intvector[10];
GLOBALREF (intvector, foo);
```

Array and structure initializers will also break the macros; you can define the initializer to be a macro of its own, or you can expand the GLOBALDEF macro by hand. You may find a case where you wish to use the GLOBALDEF macro with a large array, but you are not interested in explicitly initializing each element of the array. In such cases you can use an initializer like: {0,}, which will initialize the entire array to 0.

A shortcoming of this implementation is that a variable declared with GLOBALVALUEREF or GLOBALVALUEDEF is always an array. For example, the declaration:

```
GLOBALVALUEREF(int, ijk);
```

declares the variable ijk as an array of type int [1]. This is done because a globalvalue is actually a constant; its "value" is what the linker would normally consider an address. That is not how an integer value works in C, but it is how an array works. So treating the symbol as an array name gives consistent results—with the exception that the value seems to have the wrong type. **Don't try to access an element of the array.** It doesn't have any elements. The array "address" may not be the address of actual storage.

The fact that the symbol is an array may lead to warnings where the variable is used. Insert type casts to avoid the warnings. Here is an example; it takes advantage of the ANSI C feature allowing macros that expand to use the same name as the macro itself.

```
GLOBALVALUEREF (int, ss$_normal);
GLOBALVALUEDEF (int, xyzzy,123);
#ifdef __GNUC__
#define ss$_normal ((int) ss$_normal)
#define xyzzy ((int) xyzzy)
#endif
```

Don't use globaldef or globalref with a variable whose type is an enumeration type; this is not implemented. Instead, make the variable an integer, and use a globalvaluedef for each of the enumeration values. An example of this would be:

```
#ifdef __GNUC__
GLOBALDEF (int, color, 0);
GLOBALVALUEDEF (int, RED, 0);
GLOBALVALUEDEF (int, BLUE, 1);
GLOBALVALUEDEF (int, GREEN, 3);
#else
enum globaldef color {RED, BLUE, GREEN = 3};
#endif
```

11.3 Other VMS Issues

GCC automatically arranges for `main` to return 1 by default if you fail to specify an explicit return value. This will be interpreted by VMS as a status code indicating a normal successful completion. Version 1 of GCC did not provide this default.

GCC on VMS works only with the GNU assembler, GAS. You need version 1.37 or later of GAS in order to produce value debugging information for the VMS debugger. Use the ordinary VMS linker with the object files produced by GAS.

Under previous versions of GCC, the generated code would occasionally give strange results when linked to the sharable '`VAXCRTL`' library. Now this should work.

A caveat for use of `const` global variables: the `const` modifier must be specified in every external declaration of the variable in all of the source files that use that variable. Otherwise the linker will issue warnings about conflicting attributes for the variable. Your program will still work despite the warnings, but the variable will be placed in writable storage.

Although the VMS linker does distinguish between upper and lower case letters in global symbols, most VMS compilers convert all such symbols into upper case and most run-time library routines also have upper case names. To be able to reliably call such routines, GCC (by means of the assembler GAS) converts global symbols into upper case like other VMS compilers. However, since the usual practice in C is to distinguish case, GCC (via GAS) tries to preserve usual C behavior by augmenting each name that is not all lower case. This means truncating the name to at most 23 characters and then adding more characters at the end which encode the case pattern of those 23. Names which contain at least one dollar sign are an exception; they are converted directly into upper case without augmentation.

Name augmentation yields bad results for programs that use precompiled libraries (such as Xlib) which were generated by another compiler. You can use the compiler option '`/NOCASE_HACK`' to inhibit augmentation; it makes external C functions and variables case-independent as is usual on VMS. Alternatively, you could write all references to the functions and variables in such libraries using lower case; this will work on VMS, but is not portable to other systems. The compiler option '`/NAMES`' also provides control over global name handling.

Function and variable names are handled somewhat differently with GNU C++. The GNU C++ compiler performs *name mangling* on function names, which means that it adds information to the function name to describe the data types of the arguments that the function takes. One result of this is that the name of a function can become very long. Since the VMS linker only recognizes the first 31 characters in a name, special action is taken to ensure that each function and variable has a unique name that can be represented in 31 characters.

If the name (plus a name augmentation, if required) is less than 32 characters in length, then no special action is performed. If the name is longer than 31 characters, the assembler (GAS) will generate a hash string based upon the function name, truncate the function name to 23 characters, and append the hash string to the truncated name. If the '`/VERBOSE`' compiler option is used, the assembler will print both the full and truncated names of each symbol that is truncated.

The '`/NOCASE_HACK`' compiler option should not be used when you are compiling programs that use libg++. libg++ has several instances of objects (i.e. `Filebuf` and `filebuf`) which become indistinguishable in a case-insensitive environment. This leads to cases where you need to inhibit augmentation selectively (if you were using libg++ and Xlib in the same program, for example). There is no special feature for doing this, but you can get the result by defining a macro for each mixed case symbol for which you wish to inhibit augmentation. The macro should expand into the lower case equivalent of itself. For example:

```
#define StuDlyCapS studlycaps
```

These macro definitions can be placed in a header file to minimize the number of changes to your source code.

Index

!

'!' in constraint 298

#

'#' in constraint 299
# in template 291
#pragma 445, 446
#pragma implementation, implied 180
#pragma, reason for not using 156

$

$ 158

%

'%' in constraint 299
'%' in template 291

&

'&' in constraint 299

'

' 205

(

(nil) 246

*

'*' in constraint 299
* in template 292

-

-lgcc, use with -nodefaultlibs 45
-lgcc, use with -nostdlib 45
-nodefaultlibs and unresolved references 45
-nostdlib and unresolved references 45

.

.sdata/.sdata2 references (PowerPC) 72

/

// 157
'/f' in RTL dump 249
'/i' in RTL dump 249
'/s' in RTL dump 248, 250
'/u' in RTL dump 249
'/v' in RTL dump 248

=

'=' in constraint 298

?

'?' in constraint 298
?: extensions 143, 144
?: side effect 144

_

'_' in variables in macros 142
__bb 391
__bb_init_func 390
__bb_init_trace_func 390, 391, 392
__bb_trace_func 391, 392
__bb_trace_ret 391, 392
__builtin_apply 142
__builtin_apply_args 141
__builtin_args_info 392
__builtin_classify_type 393
__builtin_constant_p 175
__builtin_frame_address 175
__builtin_next_arg 393
__builtin_return 142
__builtin_return_address 174
__builtin_saveregs 392
__CTOR_LIST__ 423
__DTOR_LIST__ 423
__extension__ 173
__main 134

+

'+' in constraint 298

>

'>' in constraint 294
>? 178

\

\ 291

<

'<' in constraint 294
<? 178

0

'0' in constraint 295

A

abort 13, 235
abs 13, 263
abs and attributes 331
absm2 instruction pattern 308
absolute value 263
access to operands 247
accessors 247
ACCUMULATE_OUTGOING_ARGS 378
ACCUMULATE_OUTGOING_ARGS and stack frames 387
ADDITIONAL_REGISTER_NAMES 426
addm3 instruction pattern 307
addr_diff_vec 271
addr_diff_vec, length of 335
addr_vec 271
addr_vec, length of 335
address 290
address constraints 295
address of a label 138
ADDRESS_COST 406
address_operand 295
addressing modes 400
addressof 261
ADJUST_COST 409
ADJUST_FIELD_ALIGN 352
ADJUST_INSN_LENGTH 335
ADJUST_PRIORITY 409
aggregates as return values 384
alias attribute 154
aliasing of parameters 97
aligned attribute 158, 162
alignment 158
ALL_REGS 365
Alliant 201
alloca 13
alloca and SunOS 109
alloca vs variable-length arrays 146
alloca, for SunOS 131
alloca, for Unos 119
allocate_stack instruction pattern 315
ALLOCATE_TRAMPOLINE 395
ALTER_HARD_SUBREG 361
alternate keywords 173
AMD29K options 56
analysis, data flow 241
and 263
and and attributes 331
and, canonicalization of 321
andm3 instruction pattern 307
ANSI support 12
apostrophes 205
APPLY_RESULT_SIZE 384
ARC Options 90
ARG_POINTER_CFA_OFFSET 373
ARG_POINTER_REGNUM 375
ARG_POINTER_REGNUM and virtual registers 258
arg_pointer_rtx 375
ARGS_GROW_DOWNWARD 371
argument passing 237
arguments in frame (88k) 63
arguments in registers 379
arguments on stack 377
arithmetic libraries 238
arithmetic shift 263
arithmetic simplifications 239
arithmetic, in RTL 261
ARM options 57
arrays of length zero 146
arrays of variable length 146
arrays, non-lvalue 148
ashift 263
ashift and attributes 331
ashiftrt 263
ashiftrt and attributes 331
ashlm3 instruction pattern 308
ashrm3 instruction pattern 308
asm expressions 165
ASM_APP_OFF 413
ASM_APP_ON 413
ASM_BYTE_OP 415
ASM_CLOSE_PAREN 417
ASM_COMMENT_START 413
ASM_DECLARE_FUNCTION_NAME 419
ASM_DECLARE_FUNCTION_SIZE 419
ASM_DECLARE_OBJECT_NAME 419
ASM_FILE_END 413
ASM_FILE_START 413
ASM_FINAL_SPEC 342
ASM_FINISH_DECLARE_OBJECT 420
ASM_FORMAT_PRIVATE_NAME 422
asm_fprintf 428
ASM_GENERATE_INTERNAL_LABEL 421
ASM_GLOBALIZE_LABEL 420
ASM_IDENTIFY_GCC 413
asm_input 270
ASM_NO_SKIP_IN_TEXT 432
asm_noperands 276
ASM_OPEN_PAREN 417
asm_operands, RTL sharing 282
asm_operands, usage 272
ASM_OUTPUT_ADDR_DIFF_ELT 429

ASM_OUTPUT_ADDR_VEC_ELT 429
ASM_OUTPUT_ALIGN 432
ASM_OUTPUT_ALIGNED_BSS 418
ASM_OUTPUT_ALIGNED_COMMON 417
ASM_OUTPUT_ALIGNED_DECL_COMMON 418
ASM_OUTPUT_ALIGNED_DECL_LOCAL 419
ASM_OUTPUT_ALIGNED_LOCAL 419
ASM_OUTPUT_ASCII 415
ASM_OUTPUT_BSS 418
ASM_OUTPUT_BYTE 415
ASM_OUTPUT_CASE_END 430
ASM_OUTPUT_CASE_LABEL 430
ASM_OUTPUT_CHAR 415
ASM_OUTPUT_COMMON 417
ASM_OUTPUT_CONSTRUCTOR 425
ASM_OUTPUT_DEF 422
ASM_OUTPUT_DEFINE_LABEL_DIFFERENCE_SYMBOL 422
ASM_OUTPUT_DESTRUCTOR 425
ASM_OUTPUT_DOUBLE 415
ASM_OUTPUT_DOUBLE_INT 415
ASM_OUTPUT_EH_REGION_BEG 430
ASM_OUTPUT_EH_REGION_END 430
ASM_OUTPUT_EXTERNAL 421
ASM_OUTPUT_EXTERNAL_LIBCALL 421
ASM_OUTPUT_FLOAT 415
ASM_OUTPUT_IDENT 414
ASM_OUTPUT_INT 415
ASM_OUTPUT_INTERNAL_LABEL 421
ASM_OUTPUT_LABEL 419
ASM_OUTPUT_LABELREF 421
ASM_OUTPUT_LOCAL 418
ASM_OUTPUT_LONG_DOUBLE 415
ASM_OUTPUT_MAX_SKIP_ALIGN 432
ASM_OUTPUT_MI_THUNK 389
ASM_OUTPUT_OPCODE 427
ASM_OUTPUT_POOL_EPILOGUE 416
ASM_OUTPUT_POOL_PROLOGUE 416
ASM_OUTPUT_QUADRUPLE_INT 415
ASM_OUTPUT_REG_POP 429
ASM_OUTPUT_REG_PUSH 429
ASM_OUTPUT_SECTION_NAME 414
ASM_OUTPUT_SHARED_BSS 418
ASM_OUTPUT_SHARED_COMMON 418
ASM_OUTPUT_SHARED_LOCAL 419
ASM_OUTPUT_SHORT 415
ASM_OUTPUT_SKIP 431
ASM_OUTPUT_SOURCE_FILENAME 414
ASM_OUTPUT_SOURCE_LINE 414
ASM_OUTPUT_SPECIAL_POOL_ENTRY 416
ASM_OUTPUT_WEAK_ALIAS 422
ASM_SPEC 342
ASM_STABD_OP 433
ASM_STABN_OP 434
ASM_STABS_OP 433
ASM_WEAKEN_LABEL 420
assemble_name 419
assembler format 413
assembler instructions 165
assembler instructions in RTL 272
assembler names for identifiers 170
assembler syntax, 88k 63
ASSEMBLER_DIALECT 428
assembly code, invalid 217
assigning attribute values to insns 333
asterisk in template 292
atof 438
attr 332, 333
attr_flag 332
attribute expressions 330
attribute of types 161
attribute of variables 158
attribute specifications 334
attribute specifications example 334
attributes, defining 329
autoincrement addressing, availability 235
autoincrement/decrement addressing 293
autoincrement/decrement analysis 241
automatic inline for C++ member fns 165
AVOID_CCMODE_COPIES 362

B

backslash 291
backtrace for bug reports 220
barrier 275
BASE_REG_CLASS 366
basic blocks 241
bcmp 453
b*cond* instruction pattern 311
bcopy, implicit usage 398
BIGGEST_ALIGNMENT 351
BIGGEST_FIELD_ALIGNMENT 352
Bison parser generator 103
bit fields 265
bit shift overflow (88k) 65
BITFIELD_NBYTES_LIMITED 354
BITS_BIG_ENDIAN 349
BITS_BIG_ENDIAN, effect on sign_extract 265
BITS_PER_UNIT 350
BITS_PER_WORD 350
bitwise complement 263
bitwise exclusive-or 263

bitwise inclusive-or 263
bitwise logical-and 263
BLKmode 253
BLKmode, and function return values 281
BLOCK_PROFILER 390
BLOCK_PROFILER_CODE 392
bound pointer to member function 183
BRANCH_COST 407
break_out_memory_refs 402
BSS_SECTION_ASM_OP 410
bug criteria 217
bug report mailing lists 218
bugs 217
bugs, known 193
builtin functions 13
builtin_longjmp instruction pattern 316
BUILTIN_SETJMP_FRAME_VALUE 372
builtin_setjmp_receiver instruction pattern 316
builtin_setjmp_setup instruction pattern 316
byte writes (29k) 56
byte_mode 255
BYTES_BIG_ENDIAN 349
bzero 453
bzero, implicit usage 398

C

C compilation options 3
C intermediate output, nonexistent 1
C language extensions 137
C language, traditional 13
C statements for assembler output 292
C_INCLUDE_PATH 99
c++ 11
C++ 1
C++ comments 157
C++ compilation options 3
C++ interface and implementation headers 179
C++ language extensions 177
C++ member fns, automatically inline 165
C++ misunderstandings 208
C++ named return value 177
C++ options, command line 16
C++ pragmas, effect on inlining 180
C++ runtime library 110
C++ signatures 183
C++ source file suffixes 11
C++ static data, declaring and defining 208
C++ subtype polymorphism 183
C++ type abstraction 183
call 268
call instruction pattern 311
call usage 281
call-clobbered register 359
call-saved register 359
call-used register 359
call_insn 274
call_insn and '/u' 250
CALL_INSN_FUNCTION_USAGE 274
call_pop instruction pattern 312
CALL_USED_REGISTERS 359
call_used_regs 359
call_value instruction pattern 312
call_value_pop instruction pattern 312
CALLER_SAVE_PROFITABLE 386
calling conventions 371
calling functions in RTL 281
calling functions through the function vector on the H8/300 processors 155
CAN_DEBUG_WITHOUT_FP 349
CAN_ELIMINATE 377
canonicalization of instructions 320
CANONICALIZE_COMPARISON 405
canonicalize_funcptr_for_compare instruction pattern 314
case labels in initializers 149
case ranges 150
case sensitivity and VMS 232
CASE_DROPS_THROUGH 441
CASE_VALUES_THRESHOLD 442
CASE_VECTOR_MODE 441
CASE_VECTOR_PC_RELATIVE 441
CASE_VECTOR_SHORTEN_MODE 441
casesi instruction pattern 313
cast to a union 151
casts as lvalues 143
catgets 106
CC 456
cc_status 403
CC_STATUS_MDEP 403
CC_STATUS_MDEP_INIT 403
cc0 260
cc0, RTL sharing 282
cc0_rtx 260
CC1_SPEC 342
CC1PLUS_SPEC 342
CCmode 253
CDImode 253
change_address 305
CHAR_TYPE_SIZE 356
CHECK_FLOAT_VALUE 355
check_stack instruction pattern 315

CHImode ... 253
class definitions, register ... 364
class preference constraints ... 298
CLASS_LIKELY_SPILLED_P ... 369
CLASS_MAX_NREGS ... 370
classes of RTX codes ... 246
CLEAR_INSN_CACHE ... 396
CLIB ... 456
clobber ... 268
clrstr*m* instruction pattern ... 309
cmp*m* instruction pattern ... 308
cmpstr*m* instruction pattern ... 309
code generation conventions ... 92
code generation RTL sequences ... 324
code motion ... 241
code_label ... 274
code_label and '/i' ... 250
CODE_LABEL_NUMBER ... 274
codes, RTL expression ... 245
COImode ... 253
COLLECT_EXPORT_LIST ... 453
combiner pass ... 259
command options ... 3
comments, C++ style ... 157
common subexpression elimination ... 241
COMP_TYPE_ATTRIBUTES ... 446
compare ... 261
compare, canonicalization of ... 321
comparison of signed and unsigned values, warning ... 28
compilation in a separate directory ... 126
compiler bugs, reporting ... 218
compiler compared to C++ preprocessor ... 1
compiler options, C++ ... 16
compiler passes and files ... 239
compiler version, specifying ... 47
COMPILER_PATH ... 98
complement, bitwise ... 263
complex numbers ... 145
compound expressions as lvalues ... 143
computed gotos ... 138
computing the length of an insn ... 335
cond ... 265
cond and attributes ... 331
condition code register ... 260
condition code status ... 403
condition codes ... 264
conditional expressions as lvalues ... 143
conditional expressions, extensions ... 144
CONDITIONAL_REGISTER_USAGE ... 359
conditions, in patterns ... 285
configuration file ... 451
configurations supported by GNU CC ... 111
conflicting types ... 207
const applied to function ... 151
const function attribute ... 152
CONST_CALL_P ... 250
CONST_COSTS ... 405
const_double ... 255
const_double, RTL sharing ... 282
CONST_DOUBLE_CHAIN ... 256
CONST_DOUBLE_LOW ... 256
CONST_DOUBLE_MEM ... 256
CONST_DOUBLE_OK_FOR_LETTER_P ... 370
const_int ... 255
const_int and attribute tests ... 331
const_int and attributes ... 330
const_int, RTL sharing ... 282
CONST_OK_FOR_LETTER_P ... 370
const_string ... 256
const_string and attributes ... 330
const_true_rtx ... 255
const0_rtx ... 255
CONST0_RTX ... 256
const1_rtx ... 255
CONST1_RTX ... 256
const2_rtx ... 255
CONST2_RTX ... 256
constant attributes ... 336
constant folding ... 239
constant folding and floating point ... 440
constant propagation ... 241
CONSTANT_ADDRESS_P ... 400
CONSTANT_AFTER_FUNCTION_P ... 416
CONSTANT_ALIGNMENT ... 352
CONSTANT_P ... 400
CONSTANT_POOL_ADDRESS_P ... 250
CONSTANT_POOL_BEFORE_FUNCTION ... 415
constants in constraints ... 294
constm1_rtx ... 255
constraint modifier characters ... 298
constraint, matching ... 295
constraints ... 293
constraints, machine specific ... 300
constructing calls ... 141
constructor expressions ... 149
constructor function attribute ... 154
constructors vs goto ... 179
constructors, automatic calls ... 134
constructors, output of ... 423
contributors ... 469
controlling register usage ... 360

controlling the compilation driver 341
conventions, run-time 237
conversions 265
Convex options 55
copy propagation 241
copy_rtx 402
copy_rtx_if_shared 282
core dump 217
cos 13
costs of instructions 405
COSTS_N_INSNS 405
CPLUS_INCLUDE_PATH 99
CPP_PREDEFINES 347
CPP_SPEC 341
CQImode 253
cross compilation and floating point 438
cross compiling 47
cross-compiler, installation 126
cross-jumping 243
CROSS_LIBGCC1 455
CRTSTUFF_T_CFLAGS 455
CRTSTUFF_T_CFLAGS_S 455
CSImode 253
CTImode 253
CUMULATIVE_ARGS 381
current_function_epilogue_delay_list 389
current_function_is_leaf 363
current_function_outgoing_args_size 378
current_function_pops_args 388
current_function_pretend_args_size 387
current_function_uses_only_leaf_regs 363

D

'd' in constraint 294
data flow analysis 241
DATA_ALIGNMENT 352
data_section 410
DATA_SECTION_ASM_OP 410
DBR_OUTPUT_SEQEND 428
dbr_sequence_length 428
DBX 198
DBX_BLOCKS_FUNCTION_RELATIVE 435
DBX_CONTIN_CHAR 434
DBX_CONTIN_LENGTH 434
DBX_DEBUGGING_INFO 433
DBX_FUNCTION_FIRST 435
DBX_LBRAC_FIRST 435
DBX_MEMPARM_STABS_LETTER 434
DBX_NO_XREFS 434
DBX_OUTPUT_ENUM 435
DBX_OUTPUT_FUNCTION_END 435
DBX_OUTPUT_LBRAC 435
DBX_OUTPUT_MAIN_SOURCE_DIRECTORY 437
DBX_OUTPUT_MAIN_SOURCE_FILE_END 437
DBX_OUTPUT_MAIN_SOURCE_FILENAME 437
DBX_OUTPUT_RBRAC 435
DBX_OUTPUT_SOURCE_FILENAME 437
DBX_OUTPUT_STANDARD_TYPES 435
DBX_REGISTER_NUMBER 432
DBX_REGPARM_STABS_CODE 434
DBX_REGPARM_STABS_LETTER 434
DBX_STATIC_CONST_VAR_CODE 434
DBX_STATIC_STAB_DATA_SECTION 434
DBX_TYPE_DECL_STABS_CODE 434
DBX_USE_BINCL 435
DBX_WORKING_DIRECTORY 437
DCmode 253
De Morgan's law 321
dead code 240
dead_or_set_p 322
deallocating variable length arrays 146
debug_rtx 220
DEBUG_SYMS_TEXT 433
DEBUGGER_ARG_OFFSET 433
DEBUGGER_AUTO_OFFSET 432
debugging information generation 243
debugging information options 29
debugging, 88k OCS 62
declaration scope 204
declarations inside expressions 137
declarations, RTL 267
declaring attributes of functions 151
declaring static data in C++ 208
default implementation, signature member function 184
DEFAULT_CALLER_SAVES 385
DEFAULT_GDB_EXTENSIONS 433
DEFAULT_MAIN_RETURN 447
DEFAULT_PCC_STRUCT_RETURN 385
DEFAULT_RTX_COSTS 406
DEFAULT_SHORT_ENUMS 357
DEFAULT_SIGNED_CHAR 357
DEFAULT_VTABLE_THUNKS 355
define_asm_attributes 334
define_attr 329
define_delay 337
define_expand 324
define_function_unit 338
define_insn 285
define_insn example 286
define_peephole 324
define_split 327

defining attributes and their values 329
defining jump instruction patterns 319
defining peephole optimizers 322
defining RTL sequences for code generation ... 324
defining static data in C++ 208
delay slots, defining 337
DELAY_SLOTS_FOR_EPILOGUE 388
delayed branch scheduling 243
dependencies for make as output 99
dependencies, make 43
DEPENDENCIES_OUTPUT 99
Dependent Patterns 318
destructor function attribute 154
destructors vs goto 179
destructors, output of 423
detecting '-traditional' 14
DFmode 253
dialect options 12
digits in constraint 295
DImode 252
DIR_SEPARATOR 453
directory options 46
disabling certain registers 360
dispatch table 429
div 262
div and attributes 331
DIVDI3_LIBCALL 397
divide instruction, 88k 64
division 262
div*m*3 instruction pattern 307
divmod*m*4 instruction pattern 307
DIVSI3_LIBCALL 397
DOESNT_NEED_UNWINDER 431
dollar signs in identifier names 158
DOLLARS_IN_IDENTIFIERS 447
DONE 325
DONT_REDUCE_ADDR 408
double-word arithmetic 145
DOUBLE_TYPE_SIZE 357
downward funargs 139
driver 341
DW bit (29k) 56
DWARF_DEBUGGING_INFO 437
DWARF2_DEBUGGING_INFO 437
DWARF2_FRAME_INFO 438
DWARF2_UNWIND_INFO 431
DYNAMIC_CHAIN_ADDRESS 372

E

'E' in constraint 294
earlyclobber operand 299
EASY_DIV_EXPR 442
EDOM, implicit usage 398
eh_epilogue instruction pattern 316
EH_FRAME_SECTION_ASM_OP 430
EH_TABLE_LOOKUP 431
eight bit data on the H8/300 and H8/300H ... 156
ELIGIBLE_FOR_EPILOGUE_DELAY 388
ELIMINABLE_REGS 376
empty constraints 304
EMPTY_FIELD_BOUNDARY 353
ENCODE_SECTION_INFO 411
ENCODE_SECTION_INFO and address validation
.... 401
ENCODE_SECTION_INFO usage 428
ENDFILE_SPEC 343
endianness 235
enum machine_mode 252
enum reg_class 365
environment variables 97
epilogue 386
epilogue instruction pattern 317
EPILOGUE_USES 387
eq 264
eq and attributes 331
eq_attr 331
equal 264
errno, implicit usage 398
error messages 215
escape sequences, traditional 14
exception_receiver instruction pattern 316
EXCEPTION_SECTION 430
exclamation point 298
exclusive-or, bitwise 263
EXECUTABLE_SUFFIX 453
exit 13
exit status and VMS 232
EXIT_BODY 447
EXIT_IGNORE_STACK 387
EXPAND_BUILTIN_SAVEREGS 393
expander definitions 324
explicit register variables 171
expr_list 281
expression codes 245
expressions containing statements 137
expressions, compound, as lvalues 143
expressions, conditional, as lvalues 143
expressions, constructor 149
extended asm 165
extend*mn*2 instruction pattern 310
extensible constraints 295
extensions, ?: 143, 144

extensions, C language 137
extensions, C++ language 177
extern int target_flags 347
external declaration scope 204
EXTRA_CC_MODES 404
EXTRA_CC_NAMES 404
EXTRA_CONSTRAINT 370
EXTRA_SECTION_FUNCTIONS 410
EXTRA_SECTIONS 410
EXTRA_SPECS 343
extv instruction pattern 310
extzv instruction pattern 310

F

'F' in constraint 294
fabs 13
FAIL 325
fatal signal 217
FATAL_EXIT_CODE 451
features, optional, in system conventions 347
ffs 13, 263
ffs*m*2 instruction pattern 308
file name suffix 10
file names 44
files and passes of the compiler 239
final pass 243
FINAL_PRESCAN_INSN 427
FINAL_PRESCAN_LABEL 427
FINAL_REG_PARM_STACK_SPACE 378
final_scan_insn 389
final_sequence 428
FINALIZE_PIC 412
FIRST_INSN_ADDRESS 335
FIRST_PARM_OFFSET 371
FIRST_PARM_OFFSET and virtual registers 258
FIRST_PSEUDO_REGISTER 359
FIRST_STACK_REG 364
FIRST_VIRTUAL_REGISTER 258
fix 266
fix_trunc*mn*2 instruction pattern 310
fixed register 359
FIXED_REGISTERS 359
fixed_regs 359
fix*mn*2 instruction pattern 309
FIXUNS_TRUNC_LIKE_FIX_TRUNC 442
fixuns_trunc*mn*2 instruction pattern 310
fixuns*mn*2 instruction pattern 309
flags in RTL expression 248
float 266
float as function value type 205
FLOAT_ARG_TYPE 398
float_extend 266
FLOAT_STORE_FLAG_VALUE 444
float_truncate 266
FLOAT_TYPE_SIZE 357
FLOAT_VALUE_TYPE 399
FLOAT_WORDS_BIG_ENDIAN 350
FLOAT_WORDS_BIG_ENDIAN, (lack of) effect on
  subreg 259
FLOATIFY 399
floating point and cross compilation 438
floating point precision 36, 208
float*mn*2 instruction pattern 309
floatuns*mn*2 instruction pattern 309
force_reg 305
format function attribute 153
format_arg function attribute 153
forwarding calls 141
frame layout 371
FRAME_GROWS_DOWNWARD 371
FRAME_GROWS_DOWNWARD and virtual registers .. 258
frame_pointer_needed 386
FRAME_POINTER_REGNUM 374
FRAME_POINTER_REGNUM and virtual registers .. 258
FRAME_POINTER_REQUIRED 376
frame_pointer_rtx 375
frame_related, inmem 249
fscanf, and constant strings 203
ftrunc*m*2 instruction pattern 309
function addressability on the M32R/D 156
function attributes 151
function call conventions 237
function entry and exit 386
function pointers, arithmetic 148
function prototype declarations 157
function units, for scheduling 338
function, size of pointer to 148
function-call insns 281
FUNCTION_ARG 379
FUNCTION_ARG_ADVANCE 382
FUNCTION_ARG_BOUNDARY 382
FUNCTION_ARG_CALLEE_COPIES 381
FUNCTION_ARG_PADDING 382
FUNCTION_ARG_PARTIAL_NREGS 380
FUNCTION_ARG_PASS_BY_REFERENCE 381
FUNCTION_ARG_REGNO_P 382
FUNCTION_BLOCK_PROFILER 390
FUNCTION_BLOCK_PROFILER_EXIT 391
FUNCTION_BOUNDARY 351
FUNCTION_CONVERSION_BUG 452
FUNCTION_EPILOGUE 388
FUNCTION_EPILOGUE and trampolines 395

FUNCTION_INCOMING_ARG 380
FUNCTION_MODE 445
FUNCTION_OUTGOING_VALUE 383
FUNCTION_PROFILER 389
FUNCTION_PROLOGUE 386
FUNCTION_PROLOGUE and trampolines 395
FUNCTION_VALUE 383
FUNCTION_VALUE_REGNO_P 384
functions called via pointer on the RS/6000 and PowerPC 155
functions in arbitrary sections 151
functions that are passed arguments in registers on the 386 151, 155
functions that do not pop the argument stack on the 386 151
functions that do pop the argument stack on the 386 155
functions that have no side effects 151
functions that never return 151
functions that pop the argument stack on the 386 151, 155
functions which are exported from a dll on PowerPC Windows NT 155
functions which are imported from a dll on PowerPC Windows NT 155
functions which specify exception handling on PowerPC Windows NT 155
functions with printf, scanf or strftime style arguments 151
functions, leaf 363

G

'g' in constraint 295
'G' in constraint 294
g++ 11
G++ 1
GCC 1
GCC and portability 235
GCC command options 3
GCC_EXEC_PREFIX 98
ge 264
ge and attributes 331
GEN_ERRNO_RTX 398
gencodes 240
genconfig 243
general_operand 287
GENERAL_REGS 365
generalized lvalues 143
generating assembler output 292
generating insns 286
genflags 240
genflags, crash on Sun 4 195
get_attr 331
get_attr_length 336
GET_CLASS_NARROWEST_MODE 255
GET_CODE 245
get_frame_size 376
get_insns 273
get_last_insn 273
GET_MODE 254
GET_MODE_ALIGNMENT 255
GET_MODE_BITSIZE 255
GET_MODE_CLASS 254
GET_MODE_MASK 255
GET_MODE_NAME 254
GET_MODE_NUNITS 255
GET_MODE_SIZE 254
GET_MODE_UNIT_SIZE 255
GET_MODE_WIDER_MODE 254
GET_RTX_CLASS 246
GET_RTX_FORMAT 247
GET_RTX_LENGTH 247
gettext 106
geu 264
geu and attributes 331
global common subexpression elimination 241
global offset table 94
global register after longjmp 172
global register allocation 242
global register variables 171
GLOBALDEF 230
GLOBALREF 230
GLOBALVALUEDEF 230
GLOBALVALUEREF 230
GO_IF_LEGITIMATE_ADDRESS 400
GO_IF_MODE_DEPENDENT_ADDRESS 403
goto in C++ 179
goto with computed label 138
gp-relative references (MIPS) 76
gprof 30
greater than 264
grouping options 3
gt 264
gt and attributes 331
gtu 264
gtu and attributes 331

H

'H' in constraint 294
Haifa scheduler 105
HANDLE_PRAGMA 445
HANDLE_PRAGMA_PACK_PUSH_POP 446

HANDLE_SYSV_PRAGMA 446
hard registers 257
HARD_FRAME_POINTER_REGNUM 374
HARD_REGNO_CALL_PART_CLOBBERED 359
HARD_REGNO_CALLER_SAVE_MODE 386
HARD_REGNO_MODE_OK 361
HARD_REGNO_NREGS 361
hardware models and configurations, specifying 48
HAS_INIT_SECTION 425
HAVE_ATEXIT 447
HAVE_POST_DECREMENT 400
HAVE_POST_INCREMENT 400
HAVE_PRE_DECREMENT 400
HAVE_PRE_INCREMENT 400
header files and VMS 229
hex floats 146
high 257
HImode 252
HImode, in insn 276
host makefile fragment 456
HOST_BITS_PER_CHAR 451
HOST_BITS_PER_INT 451
HOST_BITS_PER_LONG 451
HOST_BITS_PER_SHORT 451
HOST_FLOAT_FORMAT 451
HOST_FLOAT_WORDS_BIG_ENDIAN 451
HOST_WORDS_BIG_ENDIAN 451
hosted environment 13
HPPA Options 79

I

'i' in constraint 294
'I' in constraint 294
i386 Options 76
IBM RS/6000 and PowerPC Options 65
IBM RT options 72
IBM RT PC 202
identifier names, dollar signs in 158
identifiers, names in assembler code 170
identifying source, compiler (88k) 62
IEEE_FLOAT_FORMAT 355
if_then_else 265
if_then_else and attributes 330
if_then_else usage 267
immediate_operand 287
IMMEDIATE_PREFIX 428
implicit argument: return value 177
IMPLICIT_FIX_EXPR 442
implied #pragma implementation 180
in_data 410
in_struct 251
in_struct, in code_label 250
in_struct, in insn 250, 251
in_struct, in label_ref 250
in_struct, in mem 248
in_struct, in reg 249
in_struct, in subreg 249
in_text 410
include files and VMS 229
INCLUDE_DEFAULTS 346
inclusive-or, bitwise 263
INCOMING_FRAME_SP_OFFSET 372
INCOMING_REGNO 360
INCOMING_RETURN_ADDR_RTX 372
incompatibilities of GCC 203
increment operators 217
INDEX_REG_CLASS 366
indirect_jump instruction pattern 313
INIT_CUMULATIVE_ARGS 381
INIT_CUMULATIVE_INCOMING_ARGS 382
INIT_ENVIRONMENT 345
INIT_SECTION_ASM_OP 410, 425
INIT_TARGET_OPTABS 398
INITIAL_ELIMINATION_OFFSET 377
INITIAL_FRAME_POINTER_OFFSET 376
initialization routines 423
initializations in expressions 149
INITIALIZE_TRAMPOLINE 395
initializers with labeled elements 149
initializers, non-constant 148
inline automatic for C++ member fns 165
inline functions 164
inline functions, omission of 165
inline, automatic 240
inlining and C++ pragmas 180
INSERT_ATTRIBUTES 447
insn 274
insn and '/i' 251
insn and '/s' 250
insn and '/u' 250
insn attributes 329
insn canonicalization 320
insn lengths, computing 335
insn splitting 327
insn-attr.h 330
INSN_ANNULLED_BRANCH_P 250
INSN_CACHE_DEPTH 396
INSN_CACHE_LINE_WIDTH 396
INSN_CACHE_SIZE 396
INSN_CLOBBERS_REGNO_P 364
INSN_CODE 276

INSN_DELETED_P 250
INSN_FROM_TARGET_P 250
insn_list 281
INSN_REFERENCES_ARE_DELAYED 448
INSN_SETS_ARE_DELAYED 447
INSN_UID 273
insns 273
insns, generating 286
insns, recognizing 286
INSTALL 456
installation trouble 193
installing GNU CC 103
installing GNU CC on the Sun 131
installing GNU CC on VMS 131
instruction attributes 329
instruction combination 241
instruction patterns 285
instruction recognizer 243
instruction scheduling 242
instruction splitting 327
insv instruction pattern 310
INT_TYPE_SIZE 356
INTEGRATE_THRESHOLD 445
integrated 252
integrated, in insn 250
integrated, in reg 249
integrating function code 164
Intel 386 Options 76
Interdependence of Patterns 318
interface and implementation headers, C++ 179
interfacing to GCC output 237
intermediate C version, nonexistent 1
Internal Compiler Checking 105
interrupt handler functions on the H8/300 processors 156
interrupt handlers on the M32R/D 156
INTIFY 399
invalid assembly code 217
invalid input 217
INVOKE__main 425
invoking g++ 12
ior 263
ior and attributes 331
ior, canonicalization of 321
ior*m*3 instruction pattern 307
IS_ASM_LOGICAL_LINE_SEPARATOR 416
isinf 440
isnan 440
ISSUE_RATE 448

J

jump instruction patterns 319
jump instructions and set 267
jump optimization 240
jump threading 241
jump_insn 274
JUMP_LABEL 274
JUMP_TABLES_IN_TEXT_SECTION 411

K

kernel and user registers (29k) 56
keywords, alternate 173
known causes of trouble 193

L

LABEL_ALIGN 431
LABEL_ALIGN_AFTER_BARRIER 431
LABEL_NUSES 275
LABEL_OUTSIDE_LOOP_P 250
LABEL_PRESERVE_P 250
label_ref 256
label_ref and '/s' 250
label_ref, RTL sharing 282
labeled elements in initializers 149
labels as values 138
labs 13
LANG 97, 99
language dialect options 12
large bit shifts (88k) 65
large return values 384
LAST_STACK_REG 364
LAST_VIRTUAL_REGISTER 258
LC_ALL 97
LC_CTYPE 97
LC_MESSAGES 97
LD_FINI_SWITCH 425
LD_INIT_SWITCH 425
LDD_SUFFIX 426
ldexp 439
le 264
le and attributes 331
leaf functions 363
leaf_function_p 312
LEAF_REG_REMAP 363
LEAF_REGISTERS 363
left rotate 263
left shift 263
LEGITIMATE_CONSTANT_P 403
LEGITIMATE_PIC_OPERAND_P 412
LEGITIMIZE_ADDRESS 402
LEGITIMIZE_RELOAD_ADDRESS 402

length-zero arrays 146
less than 264
less than or equal 264
`leu` 264
`leu` and attributes 331
`LIB_SPEC` 343
`LIB2FUNCS_EXTRA` 455
`LIBCALL_VALUE` 383
'`libgcc.a`' 397
`LIBGCC_NEEDS_DOUBLE` 398
`LIBGCC_SPEC` 343
`LIBGCC1` 455
`LIBGCC2_CFLAGS` 455
`LIBGCC2_WORDS_BIG_ENDIAN` 349
Libraries 44
library subroutine names 397
`LIBRARY_PATH` 98
`libstdc++` 110
`LIMIT_RELOAD_CLASS` 367
link options 44
`LINK_COMMAND_SPEC` 344
`LINK_LIBGCC_SPECIAL` 344
`LINK_LIBGCC_SPECIAL_1` 344
`LINK_SPEC` 342
`LINKER_DOES_NOT_WORK_WITH_DWARF2` 438
`lo_sum` 261
load address instruction 295
`LOAD_ARGS_REVERSED` 382
`LOAD_EXTEND_OP` 442
`load_multiple` instruction pattern 306
local labels 138
local register allocation 242
local variables in macros 142
local variables, specifying registers 172
`LOCAL_ALIGNMENT` 352
`LOCAL_INCLUDE_DIR` 345
`LOCAL_LABEL_PREFIX` 428
locale 97
locale definition 99
`LOG_LINKS` 276
logical-and, bitwise 263
`long long` data types 145
`LONG_DOUBLE_TYPE_SIZE` 357
`LONG_LONG_TYPE_SIZE` 356
`LONG_TYPE_SIZE` 356
`longjmp` 172
`longjmp` and automatic variables 14, 237
`longjmp` incompatibilities 204
`longjmp` warnings 26
`LONGJMP_RESTORE_FROM_STACK` 377
loop optimization 241
`LOOP_ALIGN` 431
`lshiftrt` 263
`lshiftrt` and attributes 331
`lshr`*m*`3` instruction pattern 308
`lt` 264
`lt` and attributes 331
`ltu` 264
lvalues, generalized 143

M

'`m`' in constraint 293
M32R/D options 61
M680x0 options 49
M88k options 62
machine dependent options 48
machine description macros 341
machine descriptions 285
machine mode conversions 265
machine modes 252
machine specific constraints 300
`MACHINE_DEPENDENT_REORG` 448
`MACHINE_STATE_RESTORE` 392
`MACHINE_STATE_SAVE` 391
macro with variable arguments 147
macros containing `asm` 168
macros, inline alternative 164
macros, local labels 138
macros, local variables in 142
macros, statements in expressions 137
macros, target description 341
macros, types of arguments 142
`main` and the exit status 232
make 43
`MAKE_DECL_ONE_ONLY` (*decl*) 420
`make_safe_from` 326
makefile fragment 455
`MASK_RETURN_ADDR` 431
`match_dup` 287
`match_dup` and attributes 335
`match_insn` 290
`match_insn2` 290
`match_op_dup` 289
`match_operand` 286
`match_operand` and attributes 331
`match_operator` 288
`match_par_dup` 290
`match_parallel` 289
`match_scratch` 287
matching constraint 295
matching operands 291
math libraries 238

math, in RTL 261
MATH_LIBRARY 449
MAX_BITS_PER_WORD 350
MAX_CHAR_TYPE_SIZE 357
MAX_FIXED_MODE_SIZE 354
MAX_INT_TYPE_SIZE 356
MAX_INTEGER_COMPUTATION_MODE 449
MAX_LONG_TYPE_SIZE 356
MAX_MOVE_MAX 442
MAX_OFILE_ALIGNMENT 352
MAX_REGS_PER_ADDRESS 400
MAX_WCHAR_TYPE_SIZE 358
maximum operator 178
MAYBE_REG_PARM_STACK_SPACE 378
mcount 389
MD_CALL_PROTOTYPES 453
MD_EXEC_PREFIX 345
MD_SCHED_INIT 448
MD_SCHED_REORDER 448
MD_SCHED_VARIABLE_ISSUE 448
MD_STARTFILE_PREFIX 345
MD_STARTFILE_PREFIX_1 345
mem 261
mem and '/f' 249
mem and '/s' 248
mem and '/u' 249
mem and '/v' 248
mem, RTL sharing 282
MEM_ALIAS_SET 249
MEM_IN_STRUCT_P 248
MEM_SCALAR_P 249
MEM_VOLATILE_P 248
member fns, automatically inline 165
memcmp 13
memcpy 13
memcpy, implicit usage 398
memory model (29k) 56
memory reference, nonoffsettable 297
memory references in constraints 293
MEMORY_MOVE_COST 407
memset, implicit usage 398
MERGE_MACHINE_DECL_ATTRIBUTES 446
MERGE_MACHINE_TYPE_ATTRIBUTES 446
messages, warning 22
messages, warning and error 215
middle-operands, omitted 144
MIN_UNITS_PER_WORD 350
minimum operator 178
MINIMUM_ATOMIC_ALIGNMENT 352
minus 261
minus and attributes 331
minus, canonicalization of 321
MIPS options 73
misunderstandings in C++ 208
mktemp, and constant strings 203
MN10200 options 61
MN10300 options 61
mod 262
mod and attributes 331
MODDI3_LIBCALL 397
mode attribute 159
mode classes 253
MODE_CC 254
MODE_COMPLEX_FLOAT 254
MODE_COMPLEX_INT 254
MODE_FLOAT 254
MODE_FUNCTION 254
MODE_INT 254
MODE_PARTIAL_INT 254
MODE_RANDOM 254
MODES_TIEABLE_P 362
modifiers in constraints 298
mod*m*3 instruction pattern 307
MODSI3_LIBCALL 397
MOVE_BY_PIECES_P 408
MOVE_MAX 442
MOVE_MAX_PIECES 408
MOVE_RATIO 408
mov*m* instruction pattern 305
mov*mode*cc instruction pattern 310
movstrict*m* instruction pattern 306
movstr*m* instruction pattern 308
MULDI3_LIBCALL 397
mulhisi3 instruction pattern 307
mul*m*3 instruction pattern 307
mulqihi3 instruction pattern 307
MULSI3_LIBCALL 397
mulsidi3 instruction pattern 307
mult 262
mult and attributes 331
mult, canonicalization of 321
MULTIBYTE_CHARS 452
MULTILIB_DEFAULTS 344
MULTILIB_DIRNAMES 456
MULTILIB_EXCEPTIONS 456
MULTILIB_EXTRA_OPTS 456
MULTILIB_MATCHES 456
MULTILIB_OPTIONS 455
multiple alternative constraints 297
MULTIPLE_SYMBOL_SPACES 448
multiplication 262
multiprecision arithmetic 145

MUST_PASS_IN_STACK 380
MUST_PASS_IN_STACK, and FUNCTION_ARG 380

N

'n' in constraint 294
N_REG_CLASSES 365
name augmentation 232
named patterns and conditions 285
named return value in C++ 177
names used in assembler code 170
names, pattern 305
naming convention, implementation headers ... 180
naming types 142
Native Language Support 106
ne 264
ne and attributes 331
neg 262
neg and attributes 331
neg, canonicalization of 321
neg*m*2 instruction pattern 308
nested functions 139
nested functions, trampolines for 394
newline vs string constants 15
next_cc0_user 319
NEXT_INSN 273
NEXT_OBJC_RUNTIME 399
nil 246
NLS 106
no constraints 304
no-op move instructions 243
NO_BUILTIN_PTRDIFF_TYPE 342
NO_BUILTIN_SIZE_TYPE 342
no_check_memory_usage function attribute ... 154
NO_DBX_FUNCTION_END 436
NO_DOLLAR_IN_LABEL 447
NO_DOT_IN_LABEL 447
NO_FUNCTION_CSE 409
NO_IMPLICIT_EXTERN_C 445
no_instrument_function function attribute .. 153
NO_MD_PROTOTYPES 452
no_new_pseudos 306
NO_RECURSIVE_FUNCTION_CSE 409
NO_REGS 365
NO_SYS_SIGLIST 452
nocommon attribute 160
non-constant initializers 148
non-static inline function 165
NON_SAVING_SETJMP 360
nongcc_SI_type 399
nongcc_word_type 399
nonlocal_goto instruction pattern 315
nonlocal_goto_receiver instruction pattern 316
nonoffsettable memory reference 297
nop instruction pattern 313
noreturn function attribute 151
not 263
not and attributes 331
not equal 264
not using constraints 304
not, canonicalization of 321
note 275
NOTE_INSN_BLOCK_BEG 275
NOTE_INSN_BLOCK_END 275
NOTE_INSN_DELETED 275
NOTE_INSN_EH_REGION_BEG 275
NOTE_INSN_EH_REGION_END 275
NOTE_INSN_FUNCTION_END 276
NOTE_INSN_LOOP_BEG 275
NOTE_INSN_LOOP_CONT 275
NOTE_INSN_LOOP_END 275
NOTE_INSN_LOOP_VTOP 275
NOTE_INSN_SETJMP 276
NOTE_LINE_NUMBER 275
NOTE_SOURCE_FILE 275
NOTICE_UPDATE_CC 403
NS32K options 90
NUM_MACHINE_MODES 254

O

'o' in constraint 293
OBJC_GEN_METHOD_LABEL 422
OBJC_INCLUDE_PATH 99
OBJC_INT_SELECTORS 358
OBJC_PROLOGUE 414
OBJC_SELECTORS_WITHOUT_LABELS 358
OBJECT_FORMAT_COFF 426
OBJECT_FORMAT_ROSE 426
OBJECT_SUFFIX 453
Objective C 1
Objective C threads 105
OBSTACK_CHUNK_ALLOC 451
OBSTACK_CHUNK_FREE 452
OBSTACK_CHUNK_SIZE 451
obstack_free 119
OCS (88k) 62
offsettable address 293
old-style function definitions 157
OLDAR 456
OLDCC 456
OMIT_EH_TABLE 430
omitted middle-operands 144

one_cmpl*m*2 instruction pattern . . . 308
ONLY_INT_FIELDS . . . 451
open coding . . . 164
operand access . . . 247
operand constraints . . . 293
operand substitution . . . 291
operands . . . 285
OPTIMIZATION_OPTIONS . . . 349
optimize options . . . 35
optional hardware or system features . . . 347
options to control warnings . . . 22
options, C++ . . . 16
options, code generation . . . 92
options, debugging . . . 29
options, dialect . . . 12
options, directory search . . . 46
options, GCC command . . . 3
options, grouping . . . 3
options, linking . . . 44
options, optimization . . . 35
options, order . . . 3
options, preprocessor . . . 41
order of evaluation, side effects . . . 214
order of options . . . 3
order of register allocation . . . 360
ORDER_REGS_FOR_LOCAL_ALLOC . . . 361
Ordering of Patterns . . . 317
other directory, compilation in . . . 126
OUTGOING_REG_PARM_STACK_SPACE . . . 378
OUTGOING_REGNO . . . 360
output file option . . . 11
output of assembler code . . . 413
output statements . . . 292
output templates . . . 291
output_addr_const . . . 415
output_asm_insn . . . 292
OUTPUT_QUOTED_STRING . . . 414
overflow while constant folding . . . 440
OVERLAPPING_REGNO_P . . . 364
overloaded virtual fn, warning . . . 21
OVERRIDE_OPTIONS . . . 348

P

'p' in constraint . . . 295
packed attribute . . . 160
parallel . . . 269
parameter forward declaration . . . 147
parameters, aliased . . . 97
parameters, miscellaneous . . . 441
PARM_BOUNDARY . . . 351
PARSE_LDD_OUTPUT . . . 426
parser generator, Bison . . . 103
parsing pass . . . 239
passes and files of the compiler . . . 239
passing arguments . . . 237
PATH_SEPARATOR . . . 453
PATTERN . . . 276
pattern conditions . . . 285
pattern names . . . 305
Pattern Ordering . . . 317
patterns . . . 285
pc . . . 260
pc and attributes . . . 335
pc, RTL sharing . . . 282
pc_rtx . . . 261
PCC_BITFIELD_TYPE_MATTERS . . . 353
PCC_STATIC_STRUCT_RETURN . . . 385
PDImode . . . 252
peephole optimization . . . 243
peephole optimization, RTL representation . . . 270
peephole optimizer definitions . . . 322
percent sign . . . 291
perform_... . . . 399
PIC . . . 94, 412
PIC_OFFSET_TABLE_REG_CALL_CLOBBERED . . . 412
PIC_OFFSET_TABLE_REGNUM . . . 412
plus . . . 261
plus and attributes . . . 331
plus, canonicalization of . . . 321
pmf . . . 183
Pmode . . . 445
pointer arguments . . . 152
pointer to member function . . . 183
POINTER_SIZE . . . 350
POINTERS_EXTEND_UNSIGNED . . . 350
portability . . . 235
portions of temporary objects, pointers to . . . 209
position independent code . . . 412
POSIX . . . 452
post_dec . . . 271
post_inc . . . 271
post_modify . . . 271, 272
pragma . . . 445, 446
pragma, reason for not using . . . 156
pragmas in C++, effect on inlining . . . 180
pragmas, interface and implementation . . . 179
pragmas, warning of unknown . . . 25
pre_dec . . . 271
pre_inc . . . 271
predefined macros . . . 347
PREDICATE_CODES . . . 441
PREFERRED_DEBUGGING_TYPE . . . 433

PREFERRED_OUTPUT_RELOAD_CLASS . . . 367
PREFERRED_RELOAD_CLASS . . . 367
PREFERRED_STACK_BOUNDARY . . . 351
preprocessing numbers . . . 206
preprocessing tokens . . . 206
preprocessor options . . . 41
PRETEND_OUTGOING_VARARGS_NAMED . . . 394
prev_active_insn . . . 322
prev_cc0_setter . . . 319
PREV_INSN . . . 273
PRINT_OPERAND . . . 427
PRINT_OPERAND_ADDRESS . . . 428
PRINT_OPERAND_PUNCT_VALID_P . . . 428
probe instruction pattern . . . 315
processor selection (29k) . . . 56
product . . . 262
prof . . . 30
PROFILE_BEFORE_PROLOGUE . . . 390
profile_block_flag . . . 390, 391
profiling, code generation . . . 389
program counter . . . 261
prologue . . . 386
prologue instruction pattern . . . 317
PROMOTE_FOR_CALL_ONLY . . . 351
PROMOTE_FUNCTION_ARGS . . . 351
PROMOTE_FUNCTION_RETURN . . . 351
PROMOTE_MODE . . . 350
PROMOTE_PROTOTYPES . . . 377
promotion of formal parameters . . . 157
pseudo registers . . . 257
PSImode . . . 252
PTRDIFF_TYPE . . . 357
push address instruction . . . 295
push_reload . . . 402
PUSH_ROUNDING . . . 377
PUSH_ROUNDING, interaction with PREFERRED_STACK_BOUNDARY . . . 351
PUT_CODE . . . 245
PUT_MODE . . . 254
PUT_REG_NOTE_KIND . . . 277
PUT_SDB_... . . . 438

Q

'Q', in constraint . . . 295
QImode . . . 252
QImode, in insn . . . 276
qsort, and global register variables . . . 171
question mark . . . 298
quotient . . . 262

R

'r' in constraint . . . 294
r0-relative references (88k) . . . 63
ranges in case statements . . . 150
read-only strings . . . 203
READONLY_DATA_SECTION . . . 410
REAL_ARITHMETIC . . . 440
REAL_INFINITY . . . 440
REAL_NM_FILE_NAME . . . 426
REAL_VALUE_ATOF . . . 439
REAL_VALUE_FIX . . . 439
REAL_VALUE_FROM_INT . . . 441
REAL_VALUE_ISINF . . . 440
REAL_VALUE_ISNAN . . . 440
REAL_VALUE_LDEXP . . . 439
REAL_VALUE_NEGATE . . . 440
REAL_VALUE_RNDZINT . . . 439
REAL_VALUE_TO_DECIMAL . . . 417
REAL_VALUE_TO_INT . . . 440
REAL_VALUE_TO_TARGET_DOUBLE . . . 417
REAL_VALUE_TO_TARGET_LONG_DOUBLE . . . 417
REAL_VALUE_TO_TARGET_SINGLE . . . 417
REAL_VALUE_TRUNCATE . . . 440
REAL_VALUE_TYPE . . . 439
REAL_VALUE_UNSIGNED_FIX . . . 439
REAL_VALUE_UNSIGNED_RNDZINT . . . 439
REAL_VALUES_EQUAL . . . 439
REAL_VALUES_LESS . . . 439
recog_operand . . . 427
recognizing insns . . . 286
reg . . . 257
reg and '/i' . . . 249
reg and '/s' . . . 249
reg and '/u' . . . 249
reg and '/v' . . . 249
reg, RTL sharing . . . 282
REG_ALLOC_ORDER . . . 360
REG_BR_PRED . . . 280
REG_BR_PROB . . . 280
REG_CC_SETTER . . . 280
REG_CC_USER . . . 280
REG_CLASS_CONTENTS . . . 366
REG_CLASS_FROM_LETTER . . . 366
REG_CLASS_NAMES . . . 366
REG_DEAD . . . 277
REG_DEP_ANTI . . . 280
REG_DEP_OUTPUT . . . 280
REG_EQUAL . . . 278
REG_EQUIV . . . 278
REG_EXEC_COUNT . . . 280
REG_FRAME_RELATED_EXPR . . . 280

REG_FUNCTION_VALUE_P ... 249
REG_INC ... 277
REG_LABEL ... 278
REG_LIBCALL ... 280
REG_LOOP_TEST_P ... 249
REG_MODE_OK_FOR_BASE_P ... 401
reg_names ... 428
REG_NO_CONFLICT ... 278
REG_NONNEG ... 277
REG_NOTE_KIND ... 277
REG_NOTES ... 277
REG_OK_FOR_BASE_P ... 401
REG_OK_FOR_INDEX_P ... 401
REG_OK_STRICT ... 400
REG_PARM_STACK_SPACE ... 378
REG_PARM_STACK_SPACE, and FUNCTION_ARG ... 380
REG_RETVAL ... 279
REG_UNUSED ... 279
REG_USERVAR_P ... 249
REG_WAS_0 ... 279
register allocation ... 242
register allocation order ... 360
register allocation, stupid ... 241
register class definitions ... 364
register class preference constraints ... 298
register class preference pass ... 242
register movement ... 242
register pairs ... 361
register positions in frame (88k) ... 62, 63
Register Transfer Language (RTL) ... 245
register usage ... 359
register use analysis ... 240
register variable after longjmp ... 172
register-to-stack conversion ... 243
REGISTER_MOVE_COST ... 407
REGISTER_NAMES ... 426
register_operand ... 287
REGISTER_PREFIX ... 428
registers ... 165
registers arguments ... 379
registers for local variables ... 172
registers in constraints ... 294
registers, global allocation ... 171
registers, global variables in ... 171
REGNO_MODE_OK_FOR_BASE_P ... 366
REGNO_OK_FOR_BASE_P ... 366
REGNO_OK_FOR_INDEX_P ... 366
REGNO_REG_CLASS ... 366
regs_ever_live ... 386
relative costs ... 405
RELATIVE_PREFIX_NOT_LINKDIR ... 344
reload pass ... 259
reload_completed ... 312
reload_in instruction pattern ... 306
reload_in_progress ... 305
reload_out instruction pattern ... 306
reloading ... 242
remainder ... 262
reordering, warning ... 21
reporting bugs ... 217
representation of RTL ... 245
rest argument (in macro) ... 147
rest_of_compilation ... 239
rest_of_decl_compilation ... 239
restore_stack_block instruction pattern ... 314
restore_stack_function instruction pattern ... 314
restore_stack_nonlocal instruction pattern ... 314
return ... 268
return instruction pattern ... 312
return value of main ... 232
return value, named, in C++ ... 177
return values in registers ... 383
return, in C++ function header ... 177
RETURN_ADDR_IN_PREVIOUS_FRAME ... 372
RETURN_ADDR_RTX ... 372
RETURN_ADDRESS_POINTER_REGNUM ... 375
RETURN_IN_MEMORY ... 384
RETURN_POPS_ARGS ... 379
returning aggregate values ... 384
returning structures and unions ... 237
REVERSIBLE_CC_MODE ... 405
right rotate ... 263
right shift ... 263
rotate ... 263
rotatert ... 263
rotl*m*3 instruction pattern ... 308
rotr*m*3 instruction pattern ... 308
ROUND_TYPE_ALIGN ... 354
ROUND_TYPE_SIZE ... 354
RS/6000 and PowerPC Options ... 65
RT options ... 72
RT PC ... 202
RTL addition ... 261
RTL classes ... 246
RTL comparison ... 261
RTL comparison operations ... 263
RTL constant expression types ... 255
RTL constants ... 255
RTL declarations ... 267
RTL difference ... 261

RTL expression ... 245
RTL expressions for arithmetic ... 261
RTL format ... 246
RTL format characters ... 246
RTL function-call insns ... 281
RTL generation ... 239
RTL insn template ... 286
RTL integers ... 245
RTL memory expressions ... 257
RTL object types ... 245
RTL postdecrement ... 271
RTL postincrement ... 271
RTL predecrement ... 271
RTL preincrement ... 271
RTL register expressions ... 257
RTL representation ... 245
RTL side effect expressions ... 267
RTL strings ... 245
RTL structure sharing assumptions ... 282
RTL subtraction ... 261
RTL sum ... 261
RTL vectors ... 245
RTX (See RTL) ... 245
RTX codes, classes of ... 246
`RTX_COSTS` ... 405
`RTX_FRAME_RELATED_P` ... 250
`RTX_INTEGRATED_P` ... 250
`RTX_UNCHANGING_P` ... 249
run-time conventions ... 237
run-time options ... 92
run-time target specification ... 347

S

'`s`' in constraint ... 294
`save_stack_block` instruction pattern ... 314
`save_stack_function` instruction pattern ... 314
`save_stack_nonlocal` instruction pattern ... 314
`saveable_obstack` ... 401
scalars, returned as values ... 383
`scanf`, and constant strings ... 203
`SCCS_DIRECTIVE` ... 445
`SCHED_GROUP_P` ... 251
scheduler, experimental ... 105
scheduling, delayed branch ... 243
scheduling, instruction ... 242
`SCmode` ... 253
`s`*cond* instruction pattern ... 311
scope of a variable length array ... 146
scope of declaration ... 207
scope of external declarations ... 204
`scratch` ... 259
scratch operands ... 259
`scratch`, RTL sharing ... 282
`SDB_ALLOW_FORWARD_REFERENCES` ... 438
`SDB_ALLOW_UNKNOWN_REFERENCES` ... 438
`SDB_DEBUGGING_INFO` ... 437
`SDB_DELIM` ... 438
`SDB_GENERATE_FAKE` ... 438
search path ... 46
second include path ... 42
`SECONDARY_INPUT_RELOAD_CLASS` ... 367
`SECONDARY_MEMORY_NEEDED` ... 368
`SECONDARY_MEMORY_NEEDED_MODE` ... 369
`SECONDARY_MEMORY_NEEDED_RTX` ... 369
`SECONDARY_OUTPUT_RELOAD_CLASS` ... 367
`SECONDARY_RELOAD_CLASS` ... 367
`section` function attribute ... 154
`section` variable attribute ... 160
`SELECT_CC_MODE` ... 404
`SELECT_RTX_SECTION` ... 411
`SELECT_SECTION` ... 411
separate directory, compilation in ... 126
`sequence` ... 270
sequential consistency on 88k ... 63
`set` ... 267
`set_attr` ... 333
`set_attr_alternative` ... 333
`SET_DEFAULT_DECL_ATTRIBUTES` ... 447
`SET_DEFAULT_TYPE_ATTRIBUTES` ... 446
`SET_DEST` ... 268
`SET_SRC` ... 268
`setjmp` ... 172
`setjmp` incompatibilities ... 204
`SETUP_FRAME_ADDRESSES` ... 372
`SETUP_INCOMING_VARARGS` ... 393
`SFmode` ... 253
shared strings ... 203
shared VMS run time system ... 232
`SHARED_BSS_SECTION_ASM_OP` ... 410
`SHARED_SECTION_ASM_OP` ... 410
sharing of RTL components ... 282
shift ... 263
`SHIFT_COUNT_TRUNCATED` ... 443
`SHORT_IMMEDIATES_SIGN_EXTEND` ... 442
`SHORT_TYPE_SIZE` ... 356
`sibcall_epilogue` instruction pattern ... 317
side effect in ?: ... 144
side effects, macro argument ... 137
side effects, order of evaluation ... 214
`sign_extend` ... 266
`sign_extract` ... 265
`sign_extract`, canonicalization of ... 321

signature 183
signature in C++, advantages 184
signature member function default implementation 184
signatures, C++ 183
signed and unsigned values, comparison warning 28
signed division 262
signed maximum 262
signed minimum 262
SIGNED_CHAR_SPEC 342
SImode 252
simple constraints 293
simplifications, arithmetic 239
sin 13
SIZE_TYPE 357
sizeof 142
SLOW_BYTE_ACCESS 407
SLOW_UNALIGNED_ACCESS 408
SLOW_ZERO_EXTEND 408
SMALL_REGISTER_CLASSES 369
smaller data references 62
smaller data references (88k) 63
smaller data references (MIPS) 76
smaller data references (PowerPC) 72
smax 262
smax*m*3 instruction pattern 307
smin 262
smin*m*3 instruction pattern 307
smul*m*3_highpart instruction pattern 307
SPARC options 51
specified registers 171
specifying compiler version and target machine 47
specifying hardware config 48
specifying machine version 47
specifying registers for local variables 172
speed of instructions 405
splitting instructions 327
sqrt 13, 263
sqrt*m*2 instruction pattern 308
square root 263
sscanf, and constant strings 203
stack arguments 377
stack checks (29k) 56
stack frame layout 371
STACK_BOUNDARY 351
STACK_CHECK_BUILTIN 373
STACK_CHECK_FIXED_FRAME_SIZE 374
STACK_CHECK_MAX_FRAME_SIZE 374
STACK_CHECK_MAX_VAR_SIZE 374
STACK_CHECK_PROBE_INTERVAL 373
STACK_CHECK_PROBE_LOAD 373
STACK_CHECK_PROTECT 374
STACK_DYNAMIC_OFFSET 372
STACK_DYNAMIC_OFFSET and virtual registers .. 258
STACK_GROWS_DOWNWARD 371
STACK_PARMS_IN_REG_PARM_AREA 378
STACK_POINTER_OFFSET 371
STACK_POINTER_OFFSET and virtual registers .. 258
STACK_POINTER_REGNUM 374
STACK_POINTER_REGNUM and virtual registers .. 258
stack_pointer_rtx 375
STACK_REGS 364
STACK_SAVEAREA_MODE 354
STACK_SIZE_MODE 355
stage1 108
standard pattern names 305
STANDARD_EXEC_PREFIX 344
STANDARD_INCLUDE_COMPONENT 345
STANDARD_INCLUDE_DIR 345
STANDARD_STARTFILE_PREFIX 345
start files 128
STARTFILE_SPEC 343
STARTING_FRAME_OFFSET 371
STARTING_FRAME_OFFSET and virtual registers 258
statements inside expressions 137
static data in C++, declaring and defining 208
STATIC_CHAIN 375
STATIC_CHAIN_INCOMING 375
STATIC_CHAIN_INCOMING_REGNUM 375
STATIC_CHAIN_REGNUM 375
'stdarg.h' and register arguments 380
'stdarg.h' and RT PC 72
storage layout 349
STORE_FLAG_VALUE 443
'store_multiple' instruction pattern 307
storem bug (29k) 56
strcmp 13
strcpy 13, 352
strength-reduction 241
STRICT_ALIGNMENT 353
STRICT_ARGUMENT_NAMING 394
strict_low_part 267
strict_memory_address_p 402
string constants 203
string constants vs newline 15
STRIP_NAME_ENCODING 411
strlen 13
strlen*m* instruction pattern 309
STRUCT_VALUE 385

STRUCT_VALUE_INCOMING 385
STRUCT_VALUE_INCOMING_REGNUM 385
STRUCT_VALUE_REGNUM 385
structure passing (88k) 65
structure value address 384
STRUCTURE_SIZE_BOUNDARY 353
structures 205
structures, constructor expression 149
structures, returning 237
stupid register allocation 241
sub*m*3 instruction pattern 307
submodel options 48
subreg 258
subreg and '/s' 249
subreg and '/u' 249
subreg, in strict_low_part 267
subreg, special reload handling 259
SUBREG_PROMOTED_UNSIGNED_P 249
SUBREG_PROMOTED_VAR_P 249
SUBREG_REG 259
SUBREG_WORD 259
subscripting 148
subscripting and function values 148
subtype polymorphism, C++ 183
SUCCESS_EXIT_CODE 451
suffixes for C++ source 11
Sun installation 131
SUPPORTS_ONE_ONLY 420
SUPPORTS_WEAK 420
suppressing warnings 22
surprises in C++ 208
SVr4 63
SWITCH_CURTAILS_COMPILATION 341
SWITCH_TAKES_ARG 341
SWITCHES_NEED_SPACES 341
symbol_ref 256
symbol_ref and '/u' 250
symbol_ref and '/v' 250
symbol_ref, RTL sharing 282
SYMBOL_REF_FLAG 250
SYMBOL_REF_FLAG, in ENCODE_SECTION_INFO ... 411
SYMBOL_REF_USED 250
symbolic label 282
syntax checking 22
synthesized methods, warning 22
sys_siglist 452
SYSTEM_INCLUDE_DIR 345

T

't-*target*' 455
tablejump instruction pattern 313
tagging insns 333
tail recursion optimization 240
target description macros 341
target machine, specifying 47
target makefile fragment 455
target options 47
target specifications 347
target-parameter-dependent code 240
TARGET_BELL 358
TARGET_BS 358
TARGET_CR 358
TARGET_EDOM 398
TARGET_FF 358
TARGET_FLOAT_FORMAT 355
TARGET_MEM_FUNCTIONS 398
TARGET_NEWLINE 358
TARGET_OPTIONS 348
TARGET_SWITCHES 347
TARGET_TAB 358
TARGET_VERSION 348
TARGET_VT 358
TCmode 253
tcov 30
template instantiation 180
temporaries, lifetime of 209
termination routines 423
text_section 410
TEXT_SECTION_ASM_OP 410
TFmode 253
threads, Objective C 105
Thumb Options 60
thunks 139
TImode 253
TImode, in insn 276
tiny data section on the H8/300H 156
'tm.h' macros 341
TMPDIR 98
TMS320C3x/C4x Options 87
top level of compiler 239
traditional C language 13
TRADITIONAL_RETURN_FLOAT 383
TRAMPOLINE_ALIGNMENT 395
TRAMPOLINE_SECTION 395
TRAMPOLINE_SIZE 395
TRAMPOLINE_TEMPLATE 395
trampolines for nested functions 394
TRANSFER_FROM_TRAMPOLINE 396
TRULY_NOOP_TRUNCATION 443
truncate 266
trunc*mn*2 instruction pattern 310
tst*m* instruction pattern 308

type abstraction, C++ 183
type alignment 158
type attributes 161
typedef names as function parameters 205
typeof 142

U

udiv 262
UDIVDI3_LIBCALL 397
udiv*m*3 instruction pattern 307
udivmod*m*4 instruction pattern 308
UDIVSI3_LIBCALL 397
Ultrix calling convention 202
umax 262
umax*m*3 instruction pattern 307
umin 262
umin*m*3 instruction pattern 307
umod 262
UMODDI3_LIBCALL 398
umod*m*3 instruction pattern 307
UMODSI3_LIBCALL 397
umulhisi3 instruction pattern 307
umul*m*3_highpart instruction pattern 307
umulqihi3 instruction pattern 307
umulsidi3 instruction pattern 307
unchanging 252
unchanging, in call_insn 250
unchanging, in insn 250
unchanging, in reg and mem 249
unchanging, in subreg 249
unchanging, in symbol_ref 250
undefined behavior 217
undefined function value 217
underscores in variables in macros 142
underscores, avoiding (88k) 62
union, casting to a 151
unions 205
unions, returning 237
UNIQUE_SECTION 411
UNIQUE_SECTION_P 411
UNITS_PER_WORD 350
unknown pragmas, warning 25
UNKNOWN_FLOAT_FORMAT 355
unreachable code 240
unresolved references and -nodefaultlibs 45
unresolved references and -nostdlib 45
unshare_all_rtl 282
unsigned division 262
unsigned greater than 264
unsigned less than 264
unsigned minimum and maximum 262
unsigned_fix 266
unsigned_float 266
unspec 271
unspec_volatile 271
untyped_call instruction pattern 312
untyped_return instruction pattern 313
use 269
USE_C_ALLOCA 452
USE_LOAD_POST_DECREMENT 408
USE_LOAD_POST_INCREMENT 408
USE_LOAD_PRE_DECREMENT 409
USE_LOAD_PRE_INCREMENT 409
USE_PROTOTYPES 452
USE_STORE_POST_DECREMENT 409
USE_STORE_POST_INCREMENT 409
USE_STORE_PRE_DECREMENT 409
USE_STORE_PRE_INCREMENT 409
used 251
used, in symbol_ref 250
USER_LABEL_PREFIX 428
USG 451

V

'V' in constraint 294
V850 Options 89
VALID_MACHINE_DECL_ATTRIBUTE 446
VALID_MACHINE_TYPE_ATTRIBUTE 446
value after longjmp 172
values, returned by functions 383
varargs implementation 392
'varargs.h' and RT PC 72
variable addressability on the M32R/D 161
variable alignment 158
variable attributes 158
variable number of arguments 147
variable-length array scope 146
variable-length arrays 146
variables in specified registers 171
variables, local, in macros 142
Vax calling convention 202
VAX options 51
VAX_FLOAT_FORMAT 355
'VAXCRTL' 232
VIRTUAL_INCOMING_ARGS_REGNUM 258
VIRTUAL_OUTGOING_ARGS_REGNUM 258
VIRTUAL_STACK_DYNAMIC_REGNUM 258
VIRTUAL_STACK_VARS_REGNUM 258
VMS 451
VMS and case sensitivity 232
VMS and include files 229
VMS installation 131

void pointers, arithmetic ... 148
void, size of pointer to ... 148
VOIDmode ... 253
volatil ... 251
volatil, in insn ... 250
volatil, in mem ... 248
volatil, in reg ... 249
volatil, in symbol_ref ... 250
volatile applied to function ... 151
volatile memory references ... 251
voting between constraint alternatives ... 298

W

warning for comparison of signed and unsigned values ... 28
warning for overloaded virtual fn ... 21
warning for reordering of member initializers ... 21
warning for synthesized methods ... 22
warning for unknown pragmas ... 25
warning messages ... 22
warnings vs errors ... 215
WCHAR_TYPE ... 358
WCHAR_TYPE_SIZE ... 358
weak attribute ... 154
which_alternative ... 292
whitespace ... 205
WIDEST_HARDWARE_FP_SIZE ... 357
word_mode ... 255
WORD_REGISTER_OPERATIONS ... 442
WORD_SWITCH_TAKES_ARG ... 341
WORDS_BIG_ENDIAN ... 349
WORDS_BIG_ENDIAN, effect on subreg ... 259

X

'X' in constraint ... 295
'x-*host*' ... 456
XCmode ... 253
XCOFF_DEBUGGING_INFO ... 433
XEXP ... 247
XFmode ... 253
XINT ... 247
'xm-*machine*.h' ... 451
xor ... 263
xor, canonicalization of ... 321
xor*m*3 instruction pattern ... 307
XSTR ... 247
XVEC ... 248
XVECEXP ... 248
XVECLEN ... 248
XWINT ... 247

Z

zero division on 88k ... 64
zero-length arrays ... 146
zero_extend ... 266
zero_extend*mn*2 instruction pattern ... 310
zero_extract ... 265
zero_extract, canonicalization of ... 321

Short Contents

1 Compile C, C++, Objective C, or Fortran 1
2 GCC Command Options 3
3 Installing GNU CC 103
4 Extensions to the C Language Family 137
5 Extensions to the C++ Language 185
6 `gcov`: a Test Coverage Program 195
7 Known Causes of Trouble with GCC 201
8 Reporting Bugs 225
9 How To Get Help with GCC 233
10 Contributing to GCC Development 235
11 Using GCC on VMS 237
Index 243

Table of Contents

1 Compile C, C++, Objective C, or Fortran 1

2 GCC Command Options 3

2.1 Option Summary 3
2.2 Options Controlling the Kind of Output 10
2.3 Compiling C++ Programs 11
2.4 Options Controlling C Dialect 12
2.5 Options Controlling C++ Dialect 16
2.6 Options to Request or Suppress Warnings 22
2.7 Options for Debugging Your Program or GCC 29
2.8 Options That Control Optimization 35
2.9 Options Controlling the Preprocessor 41
2.10 Passing Options to the Assembler 43
2.11 Options for Linking 44
2.12 Options for Directory Search 46
2.13 Specifying Target Machine and Compiler Version 47
2.14 Hardware Models and Configurations 48
2.14.1 M680x0 Options 48
2.14.2 VAX Options 50
2.14.3 SPARC Options 50
2.14.4 Convex Options 54
2.14.5 AMD29K Options 55
2.14.6 ARM Options 56
2.14.7 Thumb Options 59
2.14.8 MN10200 Options 60
2.14.9 MN10300 Options 60
2.14.10 M32R/D Options 61
2.14.11 M88K Options 61
2.14.12 IBM RS/6000 and PowerPC Options 64
2.14.13 IBM RT Options 71
2.14.14 MIPS Options 72
2.14.15 Intel 386 Options 76
2.14.16 HPPA Options 79
2.14.17 Intel 960 Options 80
2.14.18 DEC Alpha Options 82
2.14.19 Clipper Options 85
2.14.20 H8/300 Options 85
2.14.21 SH Options 86
2.14.22 Options for System V 86
2.14.23 TMS320C3x/C4x Options 87
2.14.24 V850 Options 89
2.14.25 ARC Options 89
2.14.26 NS32K Options 90

2.15 Options for Code Generation Conventions 92
2.16 Environment Variables Affecting GCC 97
2.17 Running Protoize 99

3 Installing GNU CC 103

3.1 Files Created by `configure` 111
3.2 Configurations Supported by GNU CC 111
3.3 Compilation in a Separate Directory 126
3.4 Building and Installing a Cross-Compiler 126
3.4.1 Steps of Cross-Compilation 127
3.4.2 Configuring a Cross-Compiler 127
3.4.3 Tools and Libraries for a Cross-Compiler 127
3.4.4 '`libgcc.a`' and Cross-Compilers 128
3.4.5 Cross-Compilers and Header Files 129
3.4.6 Actually Building the Cross-Compiler 130
3.5 Installing GNU CC on the Sun 131
3.6 Installing GNU CC on VMS 131
3.7 `collect2` 134
3.8 Standard Header File Directories 135

4 Extensions to the C Language Family 137

4.1 Statements and Declarations in Expressions 137
4.2 Locally Declared Labels 138
4.3 Labels as Values 138
4.4 Nested Functions 139
4.5 Constructing Function Calls 141
4.6 Naming an Expression's Type 142
4.7 Referring to a Type with `typeof` 142
4.8 Generalized Lvalues 143
4.9 Conditionals with Omitted Operands 144
4.10 Double-Word Integers 145
4.11 Complex Numbers 145
4.12 Hex Floats 146
4.13 Arrays of Length Zero 146
4.14 Arrays of Variable Length 146
4.15 Macros with Variable Numbers of Arguments 147
4.16 Non-Lvalue Arrays May Have Subscripts 148
4.17 Arithmetic on `void`- and Function-Pointers 148
4.18 Non-Constant Initializers 148
4.19 Constructor Expressions 149
4.20 Labeled Elements in Initializers 149
4.21 Case Ranges 150
4.22 Cast to a Union Type 151
4.23 Declaring Attributes of Functions 151
4.24 Prototypes and Old-Style Function Definitions 157
4.25 C++ Style Comments 157
4.26 Dollar Signs in Identifier Names 158
4.27 The Character ⟨ESC⟩ in Constants 158

4.28 Inquiring on Alignment of Types or Variables 158
4.29 Specifying Attributes of Variables 158
4.30 Specifying Attributes of Types 161
4.31 An Inline Function is As Fast As a Macro 164
4.32 Assembler Instructions with C Expression Operands 165
4.32.1 i386 floating point asm operands 169
4.33 Constraints for `asm` Operands 170
4.33.1 Simple Constraints 170
4.33.2 Multiple Alternative Constraints 172
4.33.3 Constraint Modifier Characters 173
4.33.4 Constraints for Particular Machines 173
4.34 Controlling Names Used in Assembler Code 178
4.35 Variables in Specified Registers 179
4.35.1 Defining Global Register Variables 179
4.35.2 Specifying Registers for Local Variables 180
4.36 Alternate Keywords 181
4.37 Incomplete `enum` Types 182
4.38 Function Names as Strings 182
4.39 Getting the Return or Frame Address of a Function 183
4.40 Other built-in functions provided by GNU CC 183
4.41 Deprecated Features 184

5 Extensions to the C++ Language 185

5.1 Named Return Values in C++ 185
5.2 Minimum and Maximum Operators in C++ 186
5.3 `goto` and Destructors in GNU C++ 187
5.4 Declarations and Definitions in One Header 187
5.5 Where's the Template? 188
5.6 Extracting the function pointer from a bound pointer to member function 191
5.7 Type Abstraction using Signatures 191

6 `gcov`: a Test Coverage Program 195

6.1 Introduction to `gcov` 195
6.2 Invoking gcov 196
6.3 Using `gcov` with GCC Optimization 198
6.4 Brief description of `gcov` data files 198

7 Known Causes of Trouble with GCC 201
7.1 Actual Bugs We Haven't Fixed Yet 201
7.2 Installation Problems 201
7.3 Cross-Compiler Problems 206
7.4 Interoperation 206
7.5 Problems Compiling Certain Programs 210
7.6 Incompatibilities of GCC 211
7.7 Fixed Header Files 214
7.8 Standard Libraries 215
7.9 Disappointments and Misunderstandings 215
7.10 Common Misunderstandings with GNU C++ 216
7.10.1 Declare *and* Define Static Members 216
7.10.2 Temporaries May Vanish Before You Expect . . . 217
7.10.3 Implicit Copy-Assignment for Virtual Bases . . . 218
7.11 Caveats of using `protoize` 219
7.12 Certain Changes We Don't Want to Make 220
7.13 Warning Messages and Error Messages 223

8 Reporting Bugs 225
8.1 Have You Found a Bug? 225
8.2 Where to Report Bugs 226
8.3 How to Report Bugs 226
8.4 Sending Patches for GCC 230

9 How To Get Help with GCC 233

10 Contributing to GCC Development 235

11 Using GCC on VMS 237
11.1 Include Files and VMS 237
11.2 Global Declarations and VMS 238
11.3 Other VMS Issues 240

Index 243