Using GNU CC

Richard M. Stallman

Last updated 26 November 1995

for version 2.7.2

For GCC Version 2.7.2

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA
Last printed November, 1995.
Printed copies are available for $50 each.
ISBN 1-882114-66-3

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

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

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

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

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DE-

FECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Contributors to GNU CC

In addition to Richard Stallman, several people have written parts of GNU CC.

- The idea of using RTL and some of the optimization ideas came from the program PO written at the University of Arizona by Jack Davidson and Christopher Fraser. See "Register Allocation and Exhaustive Peephole Optimization", Software Practice and Experience 14 (9), Sept. 1984, 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the Vax machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Support wrote the front end for C++, as well as the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series cpu, the SPARC cpu and part of the Motorola 88000 cpu.
- Gerald Baumgartner added the signature extension to the C++ front-end.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Randy Smith finished the Sun FPA support.
- Robert Brown implemented the support for Encore 32000 systems.
- David Kashtan of SRI adapted GNU CC to VMS.
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.
- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Alain Lichnewsky ported GNU CC to the Mips cpu.
- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.

- Gary Miller ported GNU CC to Charles River Data Systems machines.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination.
- Richard Kenner and Michael Tiemann jointly developed reorg.c, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- NeXT, Inc. donated the front end that supports the Objective C language.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Mike Meissner at the Open Software Foundation finished the port to the MIPS cpu, including adding ECOFF debug support, and worked on the Intel port for the Intel 80386 cpu.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Torbjorn Granlund implemented multiply- and divide-by-constant optimization, improved long long support, and improved leaf function register allocation.
- Mike Stump implemented the support for Elxsi 64 bit CPU.
- John Wehle added the machine description for the Western Electric 32000 processor used in several 3b series machines (no relation to the National Semiconductor 32000 processor).
- Holger Teutsch provided the support for the Clipper cpu.
- Kresten Krab Thorup wrote the run time support for the Objective C language.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits.
- David Edelsohn contributed the changes to RS/6000 port to make it support the PowerPC and POWER2 architectures.
- Steve Chamberlain wrote the support for the Hitachi SH processor.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Michael K. Gschwind contributed the port to the PDP-11.

1 Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU C compiler contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

2 Protect Your Freedom—Fight "Look And Feel"

> *This section is a political message from the League for Programming Freedom to the users of GNU CC. We have included it here because the issue of interface copyright is important to the GNU project.*

Apple, Lotus, and now CDC have tried to create a new form of legal monopoly: a copyright on a user interface.

An interface is a kind of language—a set of conventions for communication between two entities, human or machine. Until a few years ago, the law seemed clear: interfaces were outside the domain of copyright, so programmers could program freely and implement whatever interface the users demanded. Imitating de-facto standard interfaces, sometimes with improvements, was standard practice in the computer field. These improvements, if accepted by the users, caught on and became the norm; in this way, much progress took place.

Computer users, and most software developers, were happy with this state of affairs. However, large companies such as Apple and Lotus would prefer a different system—one in which they can own interfaces and thereby rid themselves of all serious competitors. They hope that interface copyright will give them, in effect, monopolies on major classes of software.

Other large companies such as IBM and Digital also favor interface monopolies, for the same reason: if languages become property, they expect to own many de-facto standard languages. But Apple and Lotus are the ones who have actually sued. Apple's lawsuit was defeated, for reasons only partly related to the general issue of interface copyright.

Lotus won lawsuits against two small companies, which were thus put out of business. Then they sued Borland; they won in the trial court (no surprise, since it was the same court that had ruled for Lotus twice before), but the decision was reversed by the court of appeals, with help from the League for Programming Freedom in the form of a friend-of-the-court brief. We are now waiting to see if the Supreme Court will hear the case. If it does, the League for Programming Freedom will again submit a brief.

The battle is not over. Just this summer a company that produced a simulator for a CDC computer was shut down by a copyright lawsuit by CDC, which charged that the simulator infringed the copyright on the manuals for the computer.

If the monopolists get their way, they will hobble the software field:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to design a different way to start, stop, and steer a car.
- Users will be "locked in" to whichever interface they learn; then they will be prisoners of one supplier, who will charge a monopolistic price.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can afford to sue, they can intimidate smaller developers with threats even when they don't really have a case.
- Interface improvements will come slower, since incremental evolution through creative partial imitation will no longer occur.

If interface monopolies are accepted, other large companies are waiting to grab theirs:

- Adobe is expected to claim a monopoly on the interfaces of various popular application programs, if Lotus ultimately wins the case against Borland.
- Open Computing magazine reported a Microsoft vice president as threatening to sue people who imitate the interface of Windows.

Users invest a great deal of time and money in learning to use computer interfaces. Far more, in fact, than software developers invest in developing *and even implementing* the interfaces. Whoever can own an interface, has made its users into captives, and misappropriated their investment.

To protect our freedom from monopolies like these, a group of programmers and users have formed a grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose monopolistic practices such as interface copyright and software patents. The League calls for a return to the legal policies of the recent past, in which programmers could program freely. The League is not concerned with free software as an issue, and is not affiliated with the Free Software Foundation.

The League's activities include publicizing the issues, as is being done here, and filing friend-of-the-court briefs on behalf of defendants sued by monopolists.

The League's membership rolls include Donald Knuth, the foremost authority on algorithms, John McCarthy, inventor of Lisp, Marvin Minsky, founder of the MIT Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

Activist members are especially important, but members who have no time to give are also important. Surveys at major ACM conferences have indicated a vast majority of attendees agree with the League on both issues (interface copyrights and software patents). If just ten percent of the programmers who agree with the League join the League, we will probably triumph.

To join, or for more information, phone (617) 243-4091 or write to:

League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

You can also send electronic mail to `lpf@uunet.uu.net`.

In addition to joining the League, here are some suggestions from the League for other things you can do to protect your freedom to write programs:

- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.
- Mention that you are a League member in your ‘`.signature`’, and mention the League’s email address for inquiries.
- Ask the companies you consider working for or working with to make statements against software monopolies, and give preference to those that do.
- When employers ask you to sign contracts giving them copyright on your work, insist on a clause saying they will not claim the copyright covers imitating the interface.
- When employers ask you to sign contracts giving them patent rights, insist on clauses saying they can use these rights only defensively. Don’t rely on “company policy,” since that can change at any time; don’t rely on an individual executive’s private word, since that person may be replaced. Get a commitment just as binding as the commitment they get from you.
- Write to Congress to explain the importance of these issues.

    House Subcommittee on Intellectual Property
    2137 Rayburn Bldg
    Washington, DC 20515

    Senate Subcommittee on Patents, Trademarks and Copyrights
    United States Senate
    Washington, DC 20510

  (These committees have received lots of mail already; let’s give them even more.)

Democracy means nothing if you don’t use it. Stand up and be counted!

3 Compile C, C++, or Objective C

The C, C++, and Objective C versions of the compiler are integrated; the GNU C compiler can compile programs written in C, C++, or Objective C.

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs.

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 "GNU CC" 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 "GNU CC" 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 GNU CC 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 GNU CC 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*).

4 GNU CC Command Options

When you invoke GNU CC, 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 GNU CC 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 4.3 [Compiling C++ Programs], page 26, 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.

4.1 Option Summary

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

Overall Options

See Section 4.2 [Options Controlling the Kind of Output], page 24.

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

C Language Options

See Section 4.4 [Options Controlling C Dialect], page 27.

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

C++ Language Options

See Section 4.5 [Options Controlling C++ Dialect], page 31.

-fall-virtual -fdollars-in-identifiers -felide-constructors
-fenum-int-equiv -fexternal-templates -ffor-scope -fno-for-scope
-fhandle-signatures -fmemoize-lookups -fno-default-inline -fno-gnu-keywords
-fnonnull-objects -foperator-names -fstrict-prototype
-fthis-is-variable -nostdinc++ -traditional +e*n*

Warning Options

See Section 4.6 [Options to Request or Suppress Warnings], page 35.

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

Debugging Options

See Section 4.7 [Options for Debugging Your Program or GCC], page 42.

-a -d*letters* -fpretend-float
-g -g*level* -gcoff -gdwarf -gdwarf+
-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 4.8 [Options that Control Optimization], page 46.

-fcaller-saves -fcse-follow-jumps -fcse-skip-blocks
-fdelayed-branch -fexpensive-optimizations
-ffast-math -ffloat-store -fforce-addr -fforce-mem
-finline-functions -fkeep-inline-functions
-fno-default-inline -fno-defer-pop -fno-function-cse
-fno-inline -fno-peephole -fomit-frame-pointer

`-frerun-cse-after-loop  -fschedule-insns`
`-fschedule-insns2  -fstrength-reduce  -fthread-jumps`
`-funroll-all-loops  -funroll-loops`
`-O  -O0  -O1  -O2  -O3`

Preprocessor Options

See Section 4.9 [Options Controlling the Preprocessor], page 50.

`-A`*question*`(`*answer*`)  -C  -dD  -dM  -dN`
`-D`*macro*`[=`*defn*`]  -E  -H`
`-idirafter` *dir*
`-include` *file* `-imacros` *file*
`-iprefix` *file* `-iwithprefix` *dir*
`-iwithprefixbefore` *dir* `-isystem` *dir*
`-M  -MD  -MM  -MMD  -MG  -nostdinc  -P  -trigraphs`
`-undef  -U`*macro* `-Wp,`*option*

Assembler Option

See Section 4.10 [Passing Options to the Assembler], page 52.

`-Wa,`*option*

Linker Options

See Section 4.11 [Options for Linking], page 53.

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

Directory Options

See Section 4.12 [Options for Directory Search], page 55.

`-B`*prefix* `-I`*dir* `-I-` `-L`*dir*

Target Options

See Section 4.13 [Specifying Target Machine and Compiler Version], page 56.

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

Machine Dependent Options

See Section 4.14 [Hardware Models and Configurations], page 58.

M680x0 Options
`-m68000  -m68020  -m68020-40  -m68030  -m68040  -m68881`
`-mbitfield  -mc68000  -mc68020  -mfpa  -mnobitfield`
`-mrtd  -mshort  -msoft-float`

VAX Options
`-mg  -mgnu  -munix`

SPARC Options
`-mapp-regs  -mcypress  -mepilogue  -mflat  -mfpu  -mhard-float`

```
-mhard-quad-float  -mno-app-regs  -mno-flat  -mno-fpu
-mno-epilogue  -mno-unaligned-doubles
-msoft-float  -msoft-quad-float
-msparclite  -msupersparc  -munaligned-doubles  -mv8

SPARC V9 compilers support the following options
in addition to the above:

-mmedlow  -mmedany
-mint32  -mint64  -mlong32  -mlong64
-mno-stack-bias  -mstack-bias
```

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 -m2 -m3 -m6 -mbsd -mxopen -mno-symrename
```

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
-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
-msoft-float  -mhard-float -mmultiple -mno-multiple
-mstring -mno-string -mbit-align -mno-bit-align
-mstrict-align -mno-strict-align -mrelocatable -mno-relocatable
-mtoc -mno-toc -mtraceback -mno-traceback
-mlittle -mlittle-endian -mbig -mbig-endian
-mcall-aix -mcall-sysv -mprototype
```

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  -mlong64  -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
```

i386 Options

```
-m486  -m386 -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
```

HPPA Options

```
-mdisable-fpregs  -mdisable-indexing  -mfast-indirect-calls
-mgas  -mjump-in-delay -mlong-millicode-calls -mno-disable-fpregs
-mno-disable-indexing -mno-fast-indirect-calls -mno-gas
-mno-jump-in-delay -mno-millicode-long-calls
-mno-portable-runtime -mno-soft-float -msoft-float
-mpa-risc-1-0  -mpa-risc-1-1  -mportable-runtime -mschedule=list
```

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
```

Clipper Options

```
-mc300 -mc400
```

H8/300 Options

```
-mrelax  -mh
```

System V Options

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

Code Generation Options

See Section 4.15 [Options for Code Generation Conventions], page 84.

`-fcall-saved-`*reg* `-fcall-used-`*reg*
`-ffixed-`*reg* `-finhibit-size-directive`

```
-fno-common  -fno-ident  -fno-gnu-linker
-fpcc-struct-return  -fpic  -fPIC
-freg-struct-return  -fshared-data  -fshort-enums
-fshort-double  -fvolatile  -fvolatile-global
-fverbose-asm -fpack-struct +e0  +e1
```

4.2 Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. 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.

4.3 Compiling C++ Programs

C++ source files conventionally use one of the suffixes '`.C`', '`.cc`', '`cpp`', or '`.cxx`'; preprocessed C++ files use the suffix '`.ii`'. GNU CC 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 GNU CC with the default language set to C++, and automatically specifies linking against the GNU class library libg++. [1] On many systems, the script `g++` is also installed with the name `c++`.

---

[1] Prior to release 2 of the compiler, there was a separate `g++` compiler. That version was based on GNU CC, but not integrated with it. Versions of `g++` with a '`1.`*xx*' version number—for example, `g++` version 1.37 or 1.42—are much less reliable than the versions integrated with GCC 2. Moreover, combining G++ '`1.`*xx*' with a version 2 GCC will simply not work.

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 4.4 [Options Controlling C Dialect], page 27, for explanations of options for languages related to C. See Section 4.5 [Options Controlling C++ Dialect], page 31, for explanations of options that are meaningful only for C++ programs.

4.4 Options Controlling C Dialect

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

`-ansi`
: Support all ANSI standard C programs.

  This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, disallows '`$`' as part of identifiers, and disables recognition of C++ style '`//`' comments.

  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 4.6 [Options to Request or Suppress Warnings], page 35.

  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.

`-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 two leading underscores. 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.

`-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.
- In C++ programs, assignment to `this` is permitted with '`-traditional`'. (The option '`-fthis-is-variable`' also has this effect.)

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.

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

In preprocessing directive, the '`#`' symbol must appear as the first character of a line.

In the preprocessor, 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.)

`-traditional-cpp`
: Attempt to support some aspects of traditional C preprocessors. This includes the last five items in the table immediately above, but none of the other effects of '`-traditional`'.

`-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`'.

`-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).

4.5 Options Controlling C++ Dialect

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

```
g++ -g -felide-constructors -O -c firstClass.C
```

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

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.

`-fall-virtual`
: Treat all possible member functions as virtual, implicitly. All member functions (except for constructor functions and `new` or `delete` member operators) are treated as virtual functions of the class where they appear.

: This does not mean that all calls to these member functions will be made through the internal table of virtual functions. Under some circumstances, the compiler can determine that a call to a given virtual function can be made directly; in these cases the calls are direct in any case.

`-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.

`-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.

`-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 some target

systems but not others.) Traditional C allowed the character '`$`' to form part of identifiers. However, ANSI C and C++ forbid '`$`' in identifiers.

`-fenum-int-equiv`
: Anachronistically permit implicit conversion of `int` to enumeration types. Current C++ allows conversion of `enum` to `int`, but not the other way around.

`-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 7.5 [Wherethe Template?], page 186, for more information.

`-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 7.5 [Wherethe Template?], page 186, for more information.

`-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`'.

`-fno-implicit-templates`
: Never emit code for templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Section 7.5 [Wherethe Template?], page 186, for more information.

`-fhandle-signatures`
: Recognize the `signature` and `sigof` keywords for specifying abstract types. The default ('`-fno-handle-signatures`') is not to recognize them. See Section 7.6 [`++ Signatures-title], page 188`.

`-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. If you compile any of your code with this flag, you must compile *all* of your code with this flag (including libg++, if you use it).

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

`-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.

`-fmemoize-lookups`
`-fsave-memoized`
: Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly.

The first time the compiler must build a call to a member function (or reference to a data member), it must (1) determine whether the class implements member functions of that name; (2) resolve which member function to call (which involves figuring out what sorts of type conversions need to be made); and (3) check the visibility of the member function to the caller. All of this adds up to slower compilation. Normally, the second time a call is made to that member function (or reference to that data member), it must go through the same lengthy process again. This means that code like this:

```
cout << "This " << p << " has " << n << " legs.\n";
```

makes six passes through all three steps. By using a software cache, a "hit" significantly reduces this cost. Unfortunately, using the cache introduces another layer of mechanisms which must be implemented, and so incurs its own overhead. '`-fmemoize-lookups`' enables the software cache.

Because access privileges (visibility) to members and member functions may differ from one function context to the next, G++ may need to flush the cache. With the '`-fmemoize-lookups`' flag, the cache is flushed after every function that is compiled. The '`-fsave-memoized`' flag enables the same software cache, but when the compiler determines that the context of the last function compiled would yield the same access privileges of the next function to compile, it preserves the cache. This is most helpful when defining many member functions for the same class: with the exception of member functions which are friends of other classes, each member function has exactly the same access privileges as every other, and the cache need not be flushed.

The code that implements these flags has rotted; you should probably avoid using them.

`-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`'.

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

`-fno-nonnull-objects`
: Don't assume that a reference is initialized to refer to a valid object. Although the current C++ Working Paper prohibits null references, some old code may rely on them, and you can use '`-fno-nonnull-objects`' to turn on checking.

  At the moment, the compiler only does this checking for conversions to virtual base classes.

`-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`'.

`-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`
: 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.

  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.

`-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 libg++.)

`-traditional`
: For C++ programs (in addition to the effects that apply to both C and C++), this has the same effect as '`-fthis-is-variable`'. See Section 4.4 [Options Controlling C Dialect], page 27.

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 4.8 [Options That Control Optimization], page 46.

`-Wenum-clash`
`-Woverloaded-virtual`
`-Wtemplate-debugging`
: Warnings that apply only to C++ programs. See Section 4.6 [Options to Request or Suppress Warnings], page 35.

`+e`*n*
: Control how virtual function definitions are used, in a fashion compatible with `cfront` 1.x. See Section 4.15 [Options for Code Generation Conventions], page 84.

4.6 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 GNU CC:

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

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

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

  '`-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 6.35 [Alternate Keywords], page 178.

This option is not intended to be *useful*; it exists only to satisfy pedants who would otherwise claim that GNU CC 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 recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

`-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.

`-Wformat`
: Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

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

`-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.

`-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.

To suppress this warning for an expression, simply cast it to void. For unused variables and parameters, use the '`unused`' attribute (see Section 6.28 [Specifying Attributes of Variables], page 154).

`-Wuninitialized`
An automatic variable is used without first being initialized.

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

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

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

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{

  int x;

  switch (y)

    {

    case 1: x = 1;

      break;

    case 2: x = 4;

      break;

    case 3: x = 5;

    }
```

```
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 6.22 [Declaring Attributes of Functions], page 148.

`-Wenum-clash`
: Warn about conversion between different enumeration types. (C++ only).

`-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.

`-Wtemplate-debugging`
: When using templates in a C++ program, warn if debugging is not yet fully available (C++ only).

`-Wall`
: All of the above '`-W`' options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining '-W...' options are not implied by '-Wall' because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

`-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.
- 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 };
  ```

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

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.

- A function declared external in one block and then used after the end of the block.
- A `switch` statement has an operand of type `long`.

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

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

`-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`.

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

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

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

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

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

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

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

`-Woverloaded-virtual`
: Warn when a derived class function declaration may be an error in defining a virtual function (C++ only). In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. 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.

`-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 ='.

-Werror
: Make all warnings into errors.

4.7 Options for Debugging Your Program or GNU CC

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

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

  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+', or '-gdwarf' (see below).

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

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

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

-ggdb
: Produce debugging information in the native format (if that is 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 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 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.

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gcoff`*level*
`-gxcoff`*level*
`-gdwarf`*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.

-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:

'M' Dump all macro definitions, at the end of preprocessing, and write no output.

'N' Dump all macro names, at the end of preprocessing.

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

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

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

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

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

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

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

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

'f' Dump after flow analysis, to '*file*.flow'.

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

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

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

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

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

'J' Dump after last jump optimization, to '*file*.jump2'.

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

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

'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.

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

`-save-temps`
Store the usual "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, GNU CC 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 4.16 [Environment Variables Affecting GNU CC], page 87.

4.8 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. GNU CC 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 and function inlining. 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.

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

`-fno-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.

`-fkeep-inline-functions`
: Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function.

`-fno-function-cse`
: 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`' and '`-funroll-all-loops`'. On most machines, the '`-O`' option turns on the '`-fthread-jumps`' and '`-fdelayed-branch`' options, but specific machines may handle it differently.

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

`-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.

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

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

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

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

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

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

`-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-loop`' 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`'.

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

4.9 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 4.12 [Options for Directory Search], page 55, 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 4.16 [Environment Variables Affecting GNU CC], page 87).

`-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.

4.10 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.

4.11 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 4.2 [Options Controlling the Kind of Output], page 24.

`-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.

`-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.

  One of the standard libraries bypassed by '`-nostdlib`' and '`-nodefaultlibs`' is '`libgcc.a`', a library of internal subroutines that GNU CC uses to overcome shortcomings of particular machines, or special needs for some languages. (See section "Interfacing to GNU CC Output" in *Porting GNU CC*, 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 GNU CC library subroutines. (For example, '`__main`', used to ensure C++ constructors will be called; see Section 5.6 [collect2], page 125.)

`-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. Only a few systems support this option.

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

`-Xlinker` *option*
: Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GNU CC does not know how to recognize.

  If you want to pass an option that takes an argument, you must use '`-Xlinker`' twice, once for the option and once for the argument. For example, to pass '`-assert definitions`', you must write '`-Xlinker -assert -Xlinker definitions`'. It does not work to write '`-Xlinker "-assert definitions"`', because this passes the entire string as a single argument, which is not what the linker expects.

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

`-u` *symbol* Pretend the symbol *symbol* is undefined, to force linking of library modules to define it. You can use '`-u`' multiple times with different symbols to force loading of additional library modules.

4.12 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 *directory* 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 4.13 [Specifying Target Machine and Compiler Version], page 56).

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 4.16 [Environment Variables Affecting GNU CC], page 87.

4.13 Specifying Target Machine and Compiler Version

By default, GNU CC 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 GNU CC, 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 GNU CC 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 GNU CC as a cross-compiler.

The value to use for *machine* is the same as was specified as the machine type when configuring GNU CC 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 GNU CC to run. This is useful when multiple versions are installed. For example, *version* might be '`2.0`', meaning to run GNU CC version 2.0.

The default version, when you do not specify '`-V`', is the last version of GNU CC 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 GNU CC, 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.

4.14 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.

4.14.1 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.

`-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. If your 68040 does not have code to emulate those instructions, use '`-m68040`'.

`-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.

`-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`' option implies '`-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 and 68020 processors, but not by the 68000.

4.14.2 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.

4.14.3 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 GNU CC, 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 although debugger support is still incomplete. 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', GNU CC 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.

`-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.

In a future version of GCC, these options will very likely be renamed to '`-mcpu=cypress`' and '`-mcpu=supersparc`'.

These '`-m`' switches are supported in addition to the above on SPARC V9 processors:

`-mmedlow` Generate code for the Medium/Low code model: assume a 32 bit address space. Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

  It is very likely that a future version of GCC will rename this option.

`-mmedany` Generate code for the Medium/Anywhere code model: assume a 32 bit text segment starting at offset 0, and a 32 bit data segment starting anywhere (determined at link time). Programs are statically linked, PIC is not supported. Pointers are still 64 bits.

  It is very likely that a future version of GCC will rename this option.

`-mint64` Types long and int are 64 bits.

`-mlong32` Types long and int are 32 bits.

`-mlong64`
`-mint32` Type long is 64 bits, and type int is 32 bits.

`-mstack-bias`
`-mno-stack-bias`
: With '`-mstack-bias`', GNU CC 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.

4.14.4 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.

4.14.5 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.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. 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.

4.14.6 ARM Options

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

`-m2`
`-m3`
: These options are identical. Generate code for the ARM2 and ARM3 processors. This option is the default. You should also use this option to generate code for ARM6 processors that are running with a 26-bit program counter.

`-m6`
: Generate code for the ARM6 processor when running with a 32-bit program counter.

`-mapcs`
: 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.

`-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.

4.14.7 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 GNU CC 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 GNU CC 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 GNU CC 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, GNU CC issues no such warning.

4.14.8 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`

GNU CC 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 MPC6xx microprocessors. The PowerPC architecture defines 64-bit instructions, but they are not supported by any current processors.

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 GNU CC. Specifying the '`-mcpu=`*cpu_type*' overrides the specification of these options. We recommend you use that option rather than these.

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

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

If you specify both '`-mno-power`' and '`-mno-powerpc`', GNU CC 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 GNU CC 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; GNU CC uses that mnemonic irrespective of which of these options is specified.

PowerPC assemblers support both the old and new mnemonics, as will later POWER assemblers. Current POWER assemblers only support the old mnemonics. Specify '`-mnew-mnemonics`' if you have an assembler that supports them, otherwise specify '`-mold-mnemonics`'.

The default value of these options depends on how GNU CC was configured. 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*. By default, *cpu_type* is the target system defined when GNU CC was configured. Supported values for *cpu_type* are '`rios1`', '`rios2`', '`rsc`', '`601`', '`603`', '`604`', '`power`', '`powerpc`', '`403`', and '`common`'. '`-mcpu=power`' and '`-mcpu=powerpc`' specify generic POWER and pure PowerPC (i.e., not MPC601) architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Specifying '`-mcpu=rios1`', '`-mcpu=rios2`', '`-mcpu=rsc`', or '`-mcpu=power`' enables the '`-mpower`' option and disables the '`-mpowerpc`' option; '`-mcpu=601`' enables both the '`-mpower`' and '`-mpowerpc`' options; '`-mcpu=603`', '`-mcpu=604`', '`-mcpu=403`', and '`-mcpu=powerpc`' enable the '`-mpowerpc`' option and disable the '`-mpower`' option; '`-mcpu=common`' disables both the '`-mpower`' and '`-mpowerpc`' options.

To generate code that will operate on all members of the RS/6000 and PowerPC families, specify '`-mcpu=common`'. In that case, GNU CC 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. GNU CC assumes a generic processor model for scheduling purposes.

Specifying '`-mcpu=rios1`', '`-mcpu=rios2`', '`-mcpu=rsc`', or '`-mcpu=power`' also disables the '`new-mnemonics`' option. Specifying '`-mcpu=601`', '`-mcpu=603`', '`-mcpu=604`', '`403`', or '`-mcpu=powerpc`' also enables the '`new-mnemonics`' option.

`-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, GNU CC will allocate at least one TOC entry for each unique non-automatic variable reference in your program. GNU CC 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 GNU CC from putting floating-point constants in the TOC and '`-mno-sum-in-toc`' forces GNU CC 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 GNU CC 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 GNU CC to make only one TOC entry for every file. When you specify this option, GNU CC 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.

`-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 GNU CC 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.

`-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, anod 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.

`-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.

`-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.

`-mno-traceback`
`-mtraceback`
: On embedded PowerPC systems do not (do) generate a traceback tag before the start of the function. This tag can be used by the debugger to identify where the start of a function is.

`-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-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`'.

`-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.

4.14.9 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`'.

4.14.10 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`', '`r4000`', '`r4400`', '`r4600`', and '`r6000`'. 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 the '`-mips2`' or '`-mips3`' switches 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. This option does not change the sizes of any of the C data types.

`-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` Types long, int, and pointer are 64 bits. This works only if '`-mips3`' is also specified.

`-mlong64` Types long and pointer are 64 bits, and type int is 32 bits. This works only if '`-mips3`' is also specified.

`-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.

`-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 GNU CC. 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. This requires GNU as and GNU ld which do most of the work.

`-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`'.

`-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 it's preprocessor over user assembler files (with a '`.s`' suffix) when assembling them.

4.14.11 Intel 386 Options

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

`-m486`
`-m386` Control whether or not code is optimized for a 486 instead of an 386. Code generated for an 486 will run on a 386 and vice versa.

`-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 GNU CC. 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 GNU CC 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 GNU CC 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 GNU CC 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 6.22 [Declaring Attributes of Functions], page 148

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 6.22 [Declaring Attributes of Functions], page 148

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.

`-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.

`-malign-functions=`*num*

Align the start of functions to a 2 raised to *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.

4.14.12 HPPA Options

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

`-mpa-risc-1-0`
: Generate code for a PA 1.0 processor.

`-mpa-risc-1-1`
: Generate code for a PA 1.1 processor.

`-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.

`-mmillicode-long-calls`
: Generate code which assumes millicode routines can not be reached by the standard millicode call sequence, linker-generated long-calls, or linker-modified millicode calls. In practice this should only be needed for dynamicly linked executables with extremely large SHLIB_INFO sections.

`-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.

`-mfast-indirect-calls`
: Generate code which performs faster indirect calls. Such code is suitable for kernels and for static linking. The fast indirect call code will fail miserably if it's part of a dynamically linked executable and in the presense of nested functions.

`-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`' for 7*n*0 machines, '`7100`' for 7*n*5 machines, and '`7100`' for 7*n*2 machines. '`700`' is the default for *cpu type*.

  Note the '`7100LC`' scheduling information is incomplete and using '`7100LC`' often leads to bad schedules. For now it's probably best to use '`7100`' instead of '`7100LC`' for the 7*n*2 machines.

`-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 GNU CC, with '-msoft-float' in order for this to work.

4.14.13 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 worth-

while 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). Currently this is buggy in that '#pragma align 1' is always assumed as well, and cannot be turned off.

4.14.14 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.

4.14.15 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.

4.14.16 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.

4.14.17 Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

`-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.

4.15 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.

`-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 GNU CC-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`', GNU CC defaults to whichever convention is standard for the target. If there is no standard convention, GNU CC defaults to '`-fpcc-struct-return`', except on targets where GNU CC 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 sys-

tems, 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 GNU CC 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).

`-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.

`-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). 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, GNU CC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

The GNU assembler does not fully support PIC. Currently, you must use some other assembler in order for PIC to work. We would welcome volunteers to upgrade GAS to handle this; the first part of the job is to figure out what the assembler must do differently.

`-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 allocatable 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*.

  Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

  This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-`*reg*
: Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

  Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

  A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

  This flag 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.

+e0
+e1
: Control whether virtual function definitions in classes are used to generate code, or only to define interfaces for their callers. (C++ only).

  These options are provided for compatibility with cfront 1.x usage; the recommended alternative GNU C++ usage is in flux. See Section 7.4 [Declarations and Definitions in One Header], page 184.

  With '+e0', virtual function definitions in classes are declared extern; the declaration is used only as an interface specification, not to generate code for the virtual functions (in this compilation).

  With '+e1', G++ actually generates the code implementing virtual functions defined in the code, and makes them publicly visible.

4.16 Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as '-B', '-I' and '-L' (see Section 4.12 [Options for Directory Search], page 55). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GNU CC.

TMPDIR
: If TMPDIR is set, it specifies the directory to use for temporary files. GNU CC 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 GNU CC 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), GNU CC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with '-Bfoo/', GNU CC 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. GNU CC 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, GNU CC tries the directories thus specified when searching for special linker files, if it can't find them using GCC_EXEC_PREFIX. Linking using GNU CC 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 GNU CC 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 4.9 [Options Controlling the Preprocessor], page 50), 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.

4.17 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 GNU CC itself to scan the program and collect information about the functions it uses. So neither of these programs will work until GNU CC 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 8.11 [Caveats of using "code protoize], page 210, for more information on how to use `protoize` successfully.

5 Installing GNU CC

Here is the procedure for installing GNU CC on a Unix system. See Section 5.5 [Installing GNU CC on VMS], page 121, for VMS systems. In this section we assume you compile in the same directory that contains the source files; see Section 5.2 [Compilation in a Separate Directory], page 114, 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. Specify the host, build and target machine configurations. You do this by running the file '`configure`'.

   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 '`--build`' option; the host and target will default to be the same as the build machine. (If you are building a cross-compiler, see Section 5.3 [Building and Installing a Cross-Compiler], page 115.)

   Here is an example:

   ```
   ./configure --build=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. '`sun3-bsd`' also works, since '`configure`' knows that the only BSD variant on a Sun 3 is SunOS.

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 5.1 [Configurations Supported by GNU CC], page 99, 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.

There are four additional options you can specify independently to 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. Specifically, it inhibits the installation of `collect2`, a program which otherwise serves as a front-end for the system's linker on most configurations.

'--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.

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`'.
- 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.

4. 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 `--local-prefix` option below.
5. Specify '`--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 '`--local-prefix`' **only** if your site has a different convention (not '`/usr/local`') for where to put site-specific files.

   Do not specify '`/usr`' as the '`--local-prefix`'! The directory you use for '`--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.
6. 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`'.
7. 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.
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`' and a warning about a comparison always being zero in '`enquire.o`'. Any other compilation errors may represent bugs in the port to your machine or operating system, and should be investigated and reported (see Chapter 9 [Reporting Bugs], page 217).

Some commercial 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 5.3 [Building and Installing a Cross-Compiler], page 115.
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. Then 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 9 [Reporting 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 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.

This also copies the driver program '`xgcc`' into '`/usr/local/bin/gcc`', so that it appears in typical execution search paths.

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++, it's likely that you need to also install the libg++ distribution. It should be available from the same place where you got the GNU C distribution. Just as GNU C does not distribute a C runtime library, it also does not include a C++ run-time library. All I/O functionality, special class libraries, etc., are available in the libg++ distribution.

5.1 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, 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, 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, amigados, aos, aout, bosx, bsd, clix, coff, ctix, cxux, dgux, dynix, ebmon, ecoff, elf, esix, freebsd, hms, genix, gnu, gnu/linux, 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.

Starting with GCC 2.6.1, the MIL-STD-1750A cross configuration no longer supports the Tektronix Assembler, but instead produces output for as1750, an assembler/linker available under the GNU Public License for the 1750A. Contact *kellogg@space.otn.dasa.de* for more details on obtaining 'as1750'. 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. (VMS on the Alpha is not currently supported by GNU 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.

'`arm`' 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-*-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.

'`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-*'
: DECstations can support three different personalities: Ultrix, DEC OSF/1, and OSF/rose. 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-*-*'
: 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 two variants of this CPU, called 1.0 and 1.1, which have different machine descriptions. You must use the right one for your machine. All 7*nn* machines and 8*n*7 machines use 1.1, while all other 8*nn* machines use 1.0.

  The easiest way to handle this problem is to use 'configure hp*nnn*' or 'configure hp*nnn*-hpux', where *nnn* is the model number of the machine. Then 'configure' will figure out if the machine is a 1.0 or 1.1. Use 'uname -a' to find out the model number of your machine.

  '-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-2.3 (or later) along with GDB-4.12 (or later). These can be retrieved from all the traditional GNU ftp archive sites.

Build GAS and install the resulting binary as:

`/usr/local/lib/gcc-lib/`*configuration*`/`*gccversion*`/as`

where *configuration* is the configuration name (perhaps '`hp`*nnn*`-hpux`') and *gccversion* is the GNU CC version number. Do this *before* starting the build process, otherwise you will get errors from the HPUX assembler while building '`libgcc2.a`'. The command

```
make install-dir
```

will create the necessary directory hierarchy so you can install GAS before building GCC.

To enable debugging, configure GNU CC with the '`--with-gnu-as`' option before building.

It has been reported that GNU CC produces invalid assembly code for 1.1 machines running HP-UX 8.02 when using the HP assembler. Typically the errors look like this:

```
as: bug.s @line#15 [err#1060]

  Argument 0 or 2 in FARG upper

         - lookahead = ARGW1=FR,RTNVAL=GR
as: foo.s @line#28 [err#1060]

  Argument 0 or 2 in FARG upper

         - lookahead = ARGW1=FR
```

You can check the version of HP-UX you are running by executing the command '`uname -r`'. If you are indeed running HP-UX 8.02 on a PA and using the HP assembler then configure GCC with "hp*nnn*-hpux8.02".

'`i370-*-*`'

This port is very preliminary and has many known bugs. We hope to have a higher-quality port for this machine soon.

'`i386-*-linuxoldld`'

Use this configuration to generate a.out binaries on Linux if you do not have gas/binutils version 2.5.2 or later installed. This is an obsolete configuration.

'`i386-*-linuxaout`'

Use this configuration to generate a.out binaries on Linux. This configuration is being superseded. You must use gas/binutils version 2.5.2 or later.

'`i386-*-linux`'

Use this configuration to generate ELF binaries on Linux. 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-*-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 and LD from GNU binutils version 2.2 or later.

'i386-sequent-bsd'
: Go to the Berkeley universe before compiling. In addition, you probably need to create a file named 'string.h' containing just one line: '#include <strings.h>'.

'i386-sequent-ptx1*'
: Sequent DYNIX/ptx 1.x.

'i386-sequent-ptx2*'
: Sequent DYNIX/ptx 2.x.

'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 5.4 [Installing GNU CC on the Sun], page 121, for information on installing GNU CC on Sun systems.

'i[345]86-*-winnt3.5'
: This version requires a GAS that has not let 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 8.2 [Installation Problems], page 191, 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.

'`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@cs.utah.edu` 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-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 on line 37 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 5.4 [Installing GNU CC on the Sun], page 121, 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 8.2 [Installation Problems], page 191, 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. In addition, you probably need to create a file named '`string.h`' containing just one line: '`#include <strings.h>`'.

'`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. Also, releases of AIX 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 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.

'`powerpc-*-elf`'
'`powerpc-*-sysv4`'

PowerPC system in big endian mode, running System V.4.

This configuration is currently under development.

'`powerpc-*-eabiaix`'

Embedded PowerPC system in big endian mode with -mcall-aix selected as the default. This system is currently under development.

'`powerpc-*-eabisim`'

Embedded PowerPC system in big endian mode for use in running under the PSIM simulator. This system is currently under development.

'`powerpc-*-eabi`'

Embedded PowerPC system in big endian mode.

This configuration is currently under development.

'`powerpcle-*-elf`'
'`powerpcle-*-sysv4`'

PowerPC system in little endian mode, running System V.4.

This configuration is currently under development.

'powerpcle-*-sysv4'
: Embedded PowerPC system in little endian mode.

  This system is currently under development.

'powerpcle-*-eabisim'
: Embedded PowerPC system in little endian mode for use in running under the PSIM simulator.

  This system is currently under development.

'powerpcle-*-eabi'
: Embedded PowerPC system in little endian mode.

  This configuration is currently under development.

'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 5.4 [Installing GNU CC on the Sun], page 121, for information on installing GNU CC on Sun systems.

'vax-dec-vms'
: See Section 5.5 [Installing GNU CC on VMS], page 121, 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 5.1 [Configurations Supported by GNU CC], page 99.)

  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.

5.2 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.

5.3 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.

5.3.1 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.

5.3.2 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
```

5.3.3 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 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
```

5.3.4 '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.

5.3.5 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 '/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
```

5.3.6 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.

5.4 Installing GNU CC on the Sun

On Solaris (version 2.1), do not use the linker or other tools in '/usr/ucb' to build GNU CC. Use /usr/ccs/bin.

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"
```

5.5 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 libg++, this is where the libg++ install procedure will install the libg++ 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.

5.6 collect2

Many target systems do not have support in the assembler and linker for "constructors"—initialization functions to be called before the official "start" of main. On such systems, GNU CC uses a utility called collect2 to arrange to call these 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`.

5.7 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).

`GPLUS_INCLUDE_DIR` means the same thing for native and cross. It is where `g++` looks first for header files. `libg++` 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.

6 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 7 [Extensions to the C++ Language], page 181, for extensions that apply *only* to C++.

6.1 Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type `void`, and thus effectively no value.)

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here 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 6.7 [Referring to a Type with "code typeof], page 135) or type naming (see Section 6.6 [Naming an ExpressionType], page 134).

6.2 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;                                          \
})
```

6.3 Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '`&&`'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement[1], `goto *`*exp*`;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the problem does not fit a `switch` statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You 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.

6.4 Nested Functions

A *nested function* is a function defined inside another function. (Nested functions are not supported for GNU C++.) The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, and call it twice:

[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.

```
foo (double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, here we show a nested function which uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{

  int access (int *array, int index)

    { return array[index + offset]; }

  int i;

  ...

  for (i = 0; i < size; i++)

    ... access (array, i) ...
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, before the first statement in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size)
{

  void store (int index, int value)

    { array[index] = value; }


  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit.

If you try to call the nested function through its address after the containing function has exited, all hell will break loose. If you try to call it after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

GNU CC implements taking the address of a nested function using a technique called *trampolines*. A paper describing them is available from '`maya.idiap.ch`' in directory '`pub/tmb`', file '`usenix88-lexic.ps.Z`'.

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 6.2 [Locally Declared Labels], page 128). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well. Here is an example:

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

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

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

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

6.5 Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

`__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`.

6.6 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.

6.7 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 6.35 [Alternate Keywords], page 178.

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`.

6.8 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`.

6.9 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.

6.10 Double-Word Integers

GNU C supports data types for integers that are twice as long as `long 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.

6.11 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.

6.12 Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {

  int length;

  char contents[0];
};

{

  struct line *thisline = (struct line *)

    malloc (sizeof (struct line) + this_length);

  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

6.13 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.

6.14 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.

6.15 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];
}
```

6.16 Arithmetic on `void`- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The option '`-Wpointer-arith`' requests a warning if these extensions are used.

6.17 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 };
  ...
}
```

6.18 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.

6.19 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 6.21 [Cast to a Union Type], page 147.)

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 };
```

6.20 Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from *low* to *high*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

6.21 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 6.18 [Constructor Expressions], page 143.)

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both `x` and `y` can be cast to type `union` foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x  ≡  u.i = x
u = (union foo) y  ≡  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

6.22 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. Eight attributes, `noreturn`, `const`, `format`, `section`, `constructor`, `destructor`, `unused` and `weak` are currently defined for functions. Other attributes, including `section` are supported for variables declarations (see Section 6.28 [Specifying Attributes of Variables], page 154) and for types (see Section 6.29 [Specifying Attributes of Types], page 157).

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` or `scanf` style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)

      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter *archetype* determines how the format string is interpreted, and should be either `printf` or `scanf`. 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`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using '`-Wformat`'), so there is no need to modify the header file '`stdio.h`'.

`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.

`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.

`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.

`cdecl` On the Intel 386, the `cdecl` 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. This is useful to override the effects of the '`-mrtd`' switch.

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*.

6.23 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.  */
#if __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.

6.24 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`.

6.25 Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

On some machines, dollar signs are allowed in identifiers if you specify '`-traditional`'. On a few systems they are allowed by default, even if you do not use '`-traditional`'. But they are never allowed if you specify '`-ansi`'.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

6.26 The Character `ESC` in Constants

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

6.27 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.

6.28 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 6.22 [Declaring Attributes of Functions], page 148) and for types (see Section 6.29 [Specifying Attributes of Types], page 157).

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_copy __attribute__ ((section ("INITDATACOPY"))) = 0;

main()
{

  /* Initialize stack pointer */

  init_sp (stack + sizeof (stack));


  /* Initialize initialized data */

  memcpy (&init_data_copy, &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 argument variable which is a union, means to pass the argument in the same way that the first union member would be passed. You can also use this attribute on a typedef for a union data type; then it applies to all function arguments 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 6.22 [Declaring Attributes of Functions], page 148.

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

6.29 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 6.22 [Declaring Attributes of Functions], page 148) and for variables (see Section 6.28 [Specifying Attributes of Variables], page 154).

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.

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 fas 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.

`transparent_union`

This attribute, attached to a `union` type definition, indicates that any variable having that union type should, if passed to a function, be passed in the same way that the first union member would be passed. For example:

```
union foo
{

  char a;

  int x[2];
} __attribute__ ((transparent_union));
```

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

6.30 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 6.35 [Alternate Keywords], page 178.)

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.

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 4.5 [Options Controlling C++ Dialect], page 31.)

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.

6.31 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here 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 6.32 [Constraints for "code asm Operands], page 165).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to 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, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that 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 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 does not support input-output or read-write operands. For this reason, the constraint character '`+`', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) '`combine`' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint '`"0"`' for operand 1 says that it must occupy the same location as operand 0. A digit in 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:

```
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");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many 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 all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine `_foo` accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '&' for each output operand that may not overlap an input. See Section 6.32.3 [Constraint Modifier Characters], page 169.

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 that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an `asm` instruction from being deleted, moved significantly, or combined, by writing the keyword `volatile` after the `asm`. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile `asm` instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile `asm` instructions to remain perfectly consecutive. If you want consecutive output, use a single `asm`.

It 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 6.35 [Alternate Keywords], page 178.

6.32 Constraints for `asm` Operands

Here are specific details on what constraint letters you can use with `asm` operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

6.32.1 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.

6.32.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative.

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the '?' and '!' characters:

? Disparage slightly the alternative that the '?' appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each '?' that appears in it.

! Disparage severely the alternative that the '!' appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

6.32.3 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 written 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.

`&` 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.

6.32.4 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 6.32.1 [Simple Constraints], page 166), 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)

`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)

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

`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

`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

6.33 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.

6.34 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.

  These local variables are sometimes convenient for use with the extended `asm` feature (see Section 6.31 [Assembler Instructions with C Expression Operands], page 161), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

6.34.1 Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register `a5` would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected magically by the function call mechanism.

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

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e. in a different source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. (If you are prepared to recompile `qsort` with the same global register variable, you can solve this problem.)

If you want to recompile `qsort` or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option '`-ffixed-`*reg*'. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` will not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing will happen regardless of what `longjmp` does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as `getwd`, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

6.34.2 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 6.31 [Assembler Instructions with C Expression Operands], page 161). 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`.

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 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. I would not be surprised if excessive use of this feature leaves the compiler too few available registers to compile certain functions.

6.35 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.

6.36 Incomplete enum Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

This extension is not supported by GNU C++.

6.37 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__`.

7 Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro `__GNUC__`. You can also use `__GNUG__` to test specifically for GNU C++ (see section "Standard Predefined Macros" in *The C Preprocessor*).

7.1 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.

7.2 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 6.6 [Naming an Expression's Type], page 134). However, writing `MIN` and `MAX` as macros also forces you to use function-call notation notation for a fundamental arithmetic operation. Using GNU C++ extensions, you can write '`int min = i <? j;`' instead.

Since `<?` and `>?` are built into the compiler, they properly handle expressions with side-effects; '`int min = i++ <? j++;`' works correctly.

7.3 `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. (In ANSI C++, `goto` is restricted to targets within the current block.)

The compiler still forbids using `goto` to *enter* a scope that requires constructors.

7.4 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`', '`#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 ver-

[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`'.

sion of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with '`-fno-implement-inlines`'. If any calls were not inlined, you will get linker errors.

7.5 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.

Borland model
: Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; template instances are emitted in each translation unit that uses them, and they are collapsed together at run time. 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 member templates in the header file, since they must be seen to be compiled.

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. As individual object files are built, notes are placed in the repository to record where templates and potential type arguments were seen so that the subsequent instantiation step knows where to find them. At link time, any needed instances are generated and linked in. 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; theoretically, this should be just as transparent, but in practice it has been very difficult to build multiple programs in one directory and one program in multiple directories using Cfront. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which is magically found by the link preprocessor when a template needs to be instantiated.

Currently, g++ implements neither automatic model. In the mean time, you have three options for dealing with template instantiations:

1. 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.
2. 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 use the flag -falt-external-templates instead; this flag causes template instances to be emitted in the translation unit that implements the header where they are first instantiated, rather than the one which implements the file where the templates are defined. This header must be the same in all translation units, or things are likely to break.

   See Section 7.4 [Declarations and Definitions in One Header], page 184, for more discussion of these pragmas.
3. Explicitly instantiate all the template instances you use, and compile with -fno-implicit-templates. This is probably your best bet; it may require more knowledge of exactly which templates you are using, but it's less mysterious than the previous approach, and it doesn't require any '`#pragma`'s or other g++-specific code. You can scatter the instantiations throughout your program, you can create one big file to do all the instantiations, or you can create tiny files like

   ```
   #include "Foo.h"
   #include "Foo.cc"

   template class Foo<int>;
   ```

   for each instance you need, and create a template instantiation library from those. I'm partial to the last, but your mileage may vary. 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.

7.6 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.

8 Known Causes of Trouble with GNU CC

This section describes known problems that affect users of GNU CC. Most of these are not GNU CC 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.

8.1 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, GNU C 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.

8.2 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 5.2 [Compilation in a Separate Directory], page 114.

- If you build GNU CC 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 GNU CC.
- In previous versions of GNU CC, the `gcc` driver program looked for `as` and `ld` in various places; for example, in files beginning with '`/usr/local/lib/gcc-`'. GNU CC 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 GNU CC. The fix is to get rid of the file `real-ld` which purify installs—so that GNU CC won't try to use it.
- On Linux SLS 1.01, there is a problem with '`libc.a`': it does not contain the obstack functions. However, GNU CC 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 GNU CC 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, GNU CC 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 GNU CC with the system's compiler, do not use '`-O`'. Some versions of the system's compiler miscompile GNU CC with '`-O`'.
- Sometimes on a Sun 4 you may observe a crash in the program `genflags` or `genoutput` while building GNU CC. 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 GNU CC version 2.0.2 are commonly available, but they have a bug that shows up when compiling current versions of GNU CC: undefined symbol errors occur during assembly if you use '`-g`'.

  The solution is to compile the current version of GNU CC 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 GNU CC fully. If you did not install all optional packages when installing Solaris, you will need to verify that the packages that GNU CC 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, GNU CC needs six packages: '`SUNWarc`', '`SUNWbtool`', '`SUNWesu`', '`SUNWhea`', '`SUNWlibm`', and '`SUNWtoo`'.

  For Solaris 2.2, GNU CC needs an additional seventh package: '`SUNWsprot`'.
- On Solaris 2, trying to use the linker and other tools in '`/usr/ucb`' to install GNU CC 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 GNU CC. Once GNU CC 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, GNU CC 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 GNU CC 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. GNU CC 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 GNU CC 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 GNU CC. You must use an older version of GNU CC for bootstrapping. One indication of this problem is if you get a crash when GNU CC compiles the function `muldi3` in file 'libgcc2.c'.

  You may be able to succeed by getting GNU CC version 1, installing it, and using it to compile GNU CC version 2. The bug in the Pyramid C compiler does not seem to affect GNU CC 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 GNU CC.

  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 GNU CC 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 5.1 [Configurations Supported by GNU CC], page 99.
- 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
  ```

8.3 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.

8.4 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 GNU CC version 2. If you have trouble, get GDB version 4.4 or later.
- DBX rejects some files produced by GNU CC, 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 GNU CC unless corrected by `fixincludes`. The corrected header files go in a new directory; GNU CC searches this directory before '`/usr/include`'. If you use '`-I/usr/include`', this tells GNU CC 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, GNU CC 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 GNU CC. You must specify all three options explicitly.

- On a Sparc, GNU CC 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 GNU CC, 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 GNU CC 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 GNU CC 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 GNU CC correctly. We do not yet know why. However, GNU CC 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 GNU CC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GNU CC 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 5 [Installing GNU CC], page 93).
- 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 GNU CC, but is now exceptionally rare. If you should run into it, you can work around by making your function smaller.
- GNU CC 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. GNU CC 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 5.5 [Installing GNU CC on VMS], page 121.
- 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 GNU CC 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 GNU CC 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.

GNU CC 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 GNU CC 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/2.7.1 -lgcc -lc_s
  ```

  The first specifies where to find the library '`libgcc.a`' specified with the '`-lgcc`' option.

  GNU CC 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 GNU CC 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`'.

8.5 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 GNU CC 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 GNU CC:

  ```
  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 GNU CC:

  ```
  MALLOC=gmalloc.o
  ```

8.6 Incompatibilities of GNU CC

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.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` 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 GNU CC 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.

- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument *a* is.

  The '`-traditional`' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {

    int a, b;


    a = fun1 ();

    if (setjmp (j))

      return a;


    a = fun2 ();

    /* 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 GNU CC thinks such a problem might be possible.

  The '`-traditional`' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GNU CC. 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 '+='. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.

- GNU CC 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, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '/*...*/'. However, '-traditional' suppresses these error messages.

- 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. GNU CC 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, GNU CC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GNU CC where to pass this address.

  By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GNU CC 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. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

  You can tell GNU CC 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.

8.7 Fixed Header Files

GNU CC 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 GNU CC unless they are changed. Some have bugs, some are incompatible with ANSI C, and some depend on special features of other compilers.

Installing GNU CC 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 GNU CC are not automatically updated. The easiest way to update them is to reinstall GNU CC. (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, GNU CC 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 GNU CC's fault, but it does mean that there's nothing for us to do about them.

8.8 Standard Libraries

GNU CC 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 GNU CC 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.

8.9 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 GNU CC 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 GNU CC 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 GNU CC reports an error for code like this:

  ```
  int foo (struct mumble *);

  struct mumble { ... };

  int foo (struct mumble *x)
  { ... }
  ```

  This code really is erroneous, because the scope of `struct mumble` 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.

- GNU CC 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 GNU CC 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 GNU CC 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 systems, 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 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 4.8 [Options That Control Optimization], page 46).
- 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.

8.10 Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ANSI C++ draft standard) is also evolving. 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.

8.10.1 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.

8.10.2 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 the libg++ `String` class, that define a conversion function to type `char *` or `const char *`. 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 *);
```

In this situation, it may seem natural to write '`charfunc (strfunc ());`' based on the knowledge that class `String` has an explicit conversion to `char` pointers. However, what really happens is akin to '`charfunc (strfunc ().convert ());`', where the `convert` method is a function to do the same data conversion normally performed by a cast. Since the last use of the temporary `String` object is the call to the conversion function, the compiler may delete that object before actually calling `charfunc`. The compiler has no way of knowing that deleting the `String` object will invalidate the pointer. The pointer then points to garbage, so that by the time `charfunc` is called, it gets an invalid argument.

Code like this may run successfully under some other compilers, especially those that delete temporaries relatively late. However, the GNU C++ behavior is also standard-conforming, so if your program depends on late destruction of temporaries it is not portable.

If you think this is surprising, you should be aware that the ANSI C++ committee continues to debate the lifetime-of-temporaries problem.

For now, at least, 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);
```

8.11 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.)

8.12 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 GNU CC 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 GNU CC 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 GNU CC 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, GNU CC 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.

- 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.

  GNU CC does not delete "empty" loops because the most likely reason you would put one in a program is to have a delay. Deleting them will not make real programs run any faster, so it would be pointless.

  It would be different if optimization of a nonempty loop could produce an empty one. But this generally can't happen.

- 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.

8.13 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. GNU CC 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).

GNU CC 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 GNU CC 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 4.6 [Options to Request or Suppress Warnings], page 35, for more detail on these and related command-line options.

9 Reporting Bugs

Your bug reports play an essential role in making GNU CC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 8 [Known Causes of Trouble with GNU CC], page 191. 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 10 [How To Get Help with GNU CC], page 227.) In any case, the principal function of a bug report is to help the entire community by making the next version of GNU CC work better. Bug reports are your contribution to the maintenance of GNU CC.

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.

9.1 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 8.6 [Incompatibilities of GNU CC], page 202). 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 GNU CC 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; GNU CC 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++ compilers, your suggestions for improvement of GNU CC or GNU C++ are welcome in any case.

9.2 Where to Report Bugs

Send bug reports for GNU C to '`bug-gcc@prep.ai.mit.edu`'.

Send bug reports for GNU C++ to '`bug-g++@prep.ai.mit.edu`'. If your bug involves the C++ class library libg++, send mail to '`bug-lib-g++@prep.ai.mit.edu`'. If you're not sure, you can send the bug report to both lists.

Do not send bug reports to '`help-gcc@prep.ai.mit.edu`' or to the newsgroup '`gnu.gcc.help`'. Most users of GNU CC do not want to receive bug reports. Those that do, have asked to be on '`bug-gcc`' and/or '`bug-g++`'.

The mailing lists '`bug-gcc`' and '`bug-g++`' both have newsgroups which serve as repeaters: '`gnu.gcc.bug`' and '`gnu.g++.bug`'. Each mailing list and its newsgroup carry exactly the same messages.

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
```

9.3 How to Report Bugs

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

Please report each bug in a separate message. This makes it easier for us to track which bugs have been fixed and to forward your bugs reports to the appropriate maintainer.

Do not compress and encode any part of your bug report using programs such as '`uuencode`'. If you do so it will slow down the processing of your bug. If you must submit multiple large files, use '`shar`', which allows us to read your message without having to run any decompression programs.

To enable someone to investigate the bug, you should include all these things:

- The version of GNU CC. 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 GNU CC.
- 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`'), run your source file through the C preprocessor by doing '`gcc -E` *sourcefile* `>` *outfile*', then include the contents of *outfile* in the bug report. (When you do this, use the same '`-I`', '`-D`' or '`-U`' options that you used in actual compilation.)

  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 GNU CC 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 GNU CC 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 GNU CC or GNU C`++` 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 GNU CC.
- 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 GNU CC 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 GNU CC or GNU C`++`, 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 GNU CC 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 GNU CC 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 GNU CC 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 GNU CC 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 9.4 [Sending Patches for GNU CC], page 223, 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.

9.4 Sending Patches for GNU CC

If you would like to write bug fixes or improvements for the GNU C compiler, that is very helpful. Send suggested fixes to the bug report mailing list, `bug-gcc@prep.ai.mit.edu`.

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 GNU CC 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.

10 How To Get Help with GNU CC

If you need help installing, using or changing GNU CC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `bug-gcc@prep.ai.mit.edu`, and if that brings no response, try `help-gcc@prep.ai.mit.edu`.
- 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 GNU CC distribution.

11 Using GNU CC on VMS

Here is how to use GNU CC on VMS.

11.1 Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GNU CC 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. GNU CC 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 GNU CC. VAX-C treats this much like a standard `#include <foobar.h>` directive. That is incompatible with the ANSI C behavior implemented by GNU CC: 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`'. GNU CC 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.

GNU CC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GNU CC 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 GNU CC 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

GNU CC 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

GNU CC 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 GNU CC did not provide this default.

GNU CC 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 GNU CC, 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, GNU CC (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, GNU CC (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

M

N

O

P

Short Contents

Table of Contents