CHAPTER 9

Troubleshooting GCC

As the world's most popular set of compilers, GCC has been compiled and configured for almost every modern computer system that has the resources to execute the compilers that it provides. As a set of cross-compilers, it has also been configured for many systems that can't actually run those compilers.

GCC's C and C++ compilers are the standard compilers for these languages nowadays. The new Fortran and Java compilers are exciting, and the Ada and Objective C compilers provide stable support for these languages on almost any platform. GCC's ability to function as a cross-compiler makes it easy to compile C and C++ applications that run on one platform but generate applications targeted for another, as explained elsewhere in this book. This capability even opens up GCC to being used to generate code for systems that themselves may not have the memory or disk-space resources required to run it natively. I have used GCC to build applications for everything from minicomputers to PDAs, and have thus experienced a variety of interesting occurrences that some would call bugs or broken installations. I prefer to think of them as "learning experiences."

In this chapter, I'll pass on the fruits of these experiences. This chapter is designed to help you deal with known problems that exist in GCC when used on various platforms, platform- or implementation-specific details of GCC that may be confusing, and common usage or installation problems. Luckily, usage and installation problems, by far the most common cause of GCC problems, are the easiest to correct.

You may never need to read this chapter, which is fine with me and, I'm sure, with the GNU GCC team at the Free Software Foundation and elsewhere. However, preventative medicine is good medicine; you may find it useful to skim through this chapter before you attempt to build or install GCC. Knowing what not to do is often as important as knowing what to do in the first place—though knowing what not to do may be a larger topic. If you experience problems installing or using GCC that fall outside the information provided in this chapter, please let me know by sending me an e-mail at the address given in the introduction to this book. I'll be happy to help you however I can and will add your experience and its solution to the pool of common knowledge. Be a good GCC Samaritan—perhaps your experience can save other GCC travelers from their own train wreck.

Note This chapter is not intended to replace the release notes for the current version of GCC, which will probably always be more up-to-date than this book can be—after all, GCC is constantly evolving, while this book reflects a moment in (recent) time. This chapter highlights known issues at the time that this book was written, so that you can identify (and avoid) potentially obscure problems these issues may cause.

Coping with Known Bugs and Misfeatures

The term *misfeatures* is one of the computer industry's favorite ways of referring to what more cynical people would describe as bugs, broken code, or broken scripts. In computer terms, a *feature* is an attractive capability that is provided by a software package; a misfeature is therefore a humorous way of identifying an unattractive capability (i.e., a problem).

The release notes and any online version-specific documentation associated with any GCC release are always the best documentation for a list of known problems with a specific release of GCC. Change information about GCC releases is found at URLs such as `http://gcc.gnu.org/gcc-VERSION/changes.html`, where *VERSION* is a major version of GCC, such as 3.4, 4.0, and so on. At the time this book was written, information about all the releases in the 4.0 family was provided on a single page, `http://gcc.gnu.org/gcc-4.0/changes.html`. Information about truly ancient releases of GCC may no longer be available online. If you cannot find information about the version of GCC that you are using, perhaps it is time for an upgrade.

Aside from release notes and change information, the various GCC-related newsgroups (discussed in Chapter 10) generally contain up-to-date problem reports, suggestions, questions, and general entertainment. Within a short while after a new GCC release, the Web becomes an excellent source of information, because most of the GCC newsgroups show up on archive sites that are then indexed by Web spiders and made available through popular search engines.

A few specific issues identified in the release notes for GCC at the time this book was written are the following:

- The fixincludes script that generates GCC local versions of system header files interacts badly with automounters. If the directory containing your system header files is automounted, it may be unmounted while the fixincludes script is running. The easiest way to work around this problem is to make a local copy of `/usr/include` under another name, temporarily modify your automounter maps (configuration files) not to automount `/usr/include`, and then restart autofs (on a Linux system), or whatever other automounter daemon and control scripts you are using. If your entire `/usr` directory structure is automounted, such as on many Solaris systems, simply put together a scratch machine with a local `/usr` directory, build GCC there, and then install it. You can then use the tar application to archive the directory and install it on any system, regardless of its automount configuration.
- The fixproto script may sometimes add prototypes for the `sigsetjmp()` and `siglongjmp()` functions that reference the `jmp_buf` datatype before that type is defined. You can resolve this problem by manually editing the file containing the prototypes, placing the `typedef` in front of the prototypes.

■ **Note** The fixincludes and fixproto scripts discussed in this section are only relevant if you are building GCC and they are provided as part of the GCC source code. If you are installing a newer version of GCC from an archive, or are simply using the version of GCC that came with your system, the problems noted in these scripts are irrelevant to you.

- If you specify the `-pedantic-errors` option, GCC may incorrectly give an error message when a function name is specified in an expression involving the C comma operator. This spurious error message can safely be ignored.

Using -### to See What's Going On

Before proceeding to the discussion of possible problems and suggested solutions, it is worth calling out my favorite GCC option for debugging any GCC problem (not including syntax and logical errors in my code, which, er, I have seen a few times). This is GCC's -### option, which causes any GCC compiler to display an extremely verbose listing of what it would do when compiling a specific application. This option is very similar to the standard –v option supported by all GCC compilers, with the exception that it does not produce any binaries—it just tells you what it would have done. Using the -### option is analogous to running the make program with the –n option, and it just verbosely reports what it would have done.

Consider the following examples. First, I will compile a standard hello, world application:

```
$ gcc -o hello hello.c
$
```

No output there, which is a good thing. That's a pretty simple program to screw up. Now, let's look at the output of the same compilation command when executed by adding the -### option to the command line:

```
$ gcc -### -o hello hello.c
```

```
Using built-in specs.
Target: x86_64-unknown-linux-gnu
Configured with: ../gcc/configure --prefix=/usr/local/gcc4.2svn --enable-threads \
    --enable-languages=c,c++,objc,fortran,obj-c++
Thread model: posix
gcc version 4.2.0 20060102 (experimental)
 "/usr/local/gcc4.2svn/libexec/gcc/x86_64-unknown-linux-gnu/4.2.0/cc1" ➥
     "-quiet" "hello.c" "-quiet" "-dumpbase" "hello.c" "-mtune=k8" "-auxbase" ➥
     "hello" "-o" "/tmp/ccoBIRAk.s"
 "as" "-Qy" "-o" "/tmp/ccpqTsUl.o" "/tmp/ccoBIRAk.s"
 "/usr/local/gcc4.2svn/libexec/gcc/x86_64-unknown-linux-gnu/4.2.0/collect2" ➥
     "--eh-frame-hdr" "-m" "elf_x86_64" "-dynamic-linker" ➥
     "/lib64/ld-linux-x86-64.so.2" ➥
     "-o" "hello" "/usr/lib/../lib64/crt1.o" "/usr/lib/../lib64/crti.o" ➥
     "/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/crtbegin.o" ➥
     "-L/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0" ➥
     "-L/usr/local/gcc4.2svn/lib64" ➥
     "-L/lib/../lib64" "-L/usr/lib/../lib64" ➥
     "-L/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/../../.." ➥
     "/tmp/ccpqTsUl.o" "-lgcc" "--as-needed" "-lgcc_s" ➥
     "--no-as-needed" "-lc" ➥
     "-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed" ➥
     "/usr/local/gcc4.2svn/lib/gcc/x86_64-unknown-linux-gnu/4.2.0/crtend.o" ➥
     "/usr/lib/../lib64/crtn.o"
$
```

I have broken many of these lines so that they fit on the printed page, but I think that you get the idea. This sort of verbose output (without execution) makes it easy to spot configuration, installation, include, or library problems that can prevent any GCC compiler from working correctly. To correct some of these sorts of problems, you can simply adjust library or include file paths. For internal problems, you can dump the specs file and fix the errors in there; but in many cases, you may need to rebuild or reinstall your GCC compilers.

Resolving Common Problems

One of the biggest advantages of open source software is that everyone has access to the source code and can therefore put it under a microscope whenever necessary to identify and resolve problems. For GCC, quite probably the most popular open source package of all time, a parallel advantage is that you are rarely the first person to encounter a specific problem. The Web and the GCC documentation are excellent sources for descriptions of common problems encountered when using any GCC compiler and, more importantly, how to solve them.

This section lists some common problems you may run into when using GCC on a variety of platforms and in a variety of ways, and provides solutions or workarounds. It is important to recognize that the fact that one can identify common problems does not mean that GCC itself is laden with problems. As you can see from the number of command-line options discussed in appendixes A and B, and some of the advanced usage models discussed in the chapters on specific GCC compilers, the GCC compilers are some of the world's most flexible programs. The number of knobs that you can turn on GCC compilers sometime leads to problems by accidentally invoking conflicting options, environment variables settings, and so on.

Problems Executing GCC

It is very rare that a version of GCC is installed in a way that all users of a system cannot execute it. Installing GCC incorrectly could actually cause this problem, in which case you would see a message like the following when trying to execute the GCC C compiler:

```
bash: gcc: Permission denied
```

If you get this message, you will have to contact your system administrator to see if gcc was intentionally installed with restrictive file permissions or ACLs (access control lists).

More commonly, problems executing a GCC compiler on a system are related to making sure that you are executing the right version of the compiler, as explained in the following two sections.

Using Multiple Versions of GCC on a Single System

If you are running an open source operating system such as Linux, FreeBSD, NetBSD, or one of the others, there is a good chance that your system came with a version of GCC that is included as part of your operating system's default installation. This is incredibly useful for most day-to-day purposes, but you may eventually want to upgrade the version of GCC available on your system, often because a new version adds features or capabilities not present in an earlier version.

Having multiple versions of GCC installed on a single system is not usually a problem. As explained in Chapter 11 and Appendix A, each version of GCC is consistent and contains internal references to the location where it was intended to be installed. The GCC C compiler binary (gcc) is primarily a driver program, using internal specifications to determine the options with which to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, the GCC C compiler binary itself (gcc) currently contains hard-coded strings that identify the directory where that version of gcc expects to find these programs.

Unfortunately, gcc's internal consistency cannot guarantee that you are executing the "right" version of gcc. If you have multiple versions of the GCC compilers installed on your system and they were all installed using the default names, make sure that your `PATH` environment variable is set so

that the directory where the version you are trying to execute is listed before any directory containing another version of GCC. You can also execute any GCC compiler by using its full pathname, as in `/usr/local/gcc41/bin/gcc`. If installed correctly, each GCC binary will execute the versions of all the other programs used during the compilation process that were built at the same time as that version of gcc.

Setting the value of your `PATH` environment variable is done differently depending on the command shell that you are running. If you are using the Bourne-Again shell (bash), you can set the value of this variable by using a command such as the following:

```
$ export PATH=new-directory:${PATH}
```

Replace `new-directory` with the full pathname of the directory containing the version of GCC that you actually mean to run, as in the following example, which puts the directory `/usr/local/bin` at the beginning of your existing search `PATH`:

```
$ export PATH=/usr/local/bin:${PATH}
```

If you are using the C shell (csh) or the TOPS-20/TENEX C shell (tcsh), the syntax for updating the `PATH` environment variable is slightly different.

```
% setenv PATH /usr/local/bin:${PATH}
```

■ **Tip** On most Linux and other Unix-like systems, you can find all versions of GCC that are currently present in any directory in your `PATH` by executing the `whereis gcc` command, which will return something like the following:

```
$ whereis gcc
gcc: /usr/bin/gcc /usr/local/tivo-mips/bin/gcc /usr/local/bin/gcc
```

You can then use the `which gcc` command to identify which one is located first in your `PATH`, and adjust your `PATH` as necessary.

Problems Loading Libraries When Executing Programs

Encountering messages such as the following when trying to execute a program that you just built is a common source of frustration:

```
/foo: error while loading shared libraries: libfoo-0.9.so.21: ➥
    cannot open shared object file: No such file or directory
```

This isn't really GCC's fault—instead, what this means is that the application uses shared libraries and the loader can't locate a shared library that the application requires. The most common solution for this is to investigate where the specified library is installed, and to make sure that this directory is listed in the text file /etc/ld.so.conf. This file contains a list of the directories that the loader should search for shared libraries—one directory per line. If you have to add a new directory to this file, you must also run the `ldconfig` command after updating /etc/ld.so.conf to ensure that the loader's in-memory cache of where to look for shared libraries has been updated.

Most applications that build shared libraries display a message stating where they have been installed, as in the following example:

```
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/somelib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
```

In this case, you would need to add the directory `/usr/local/somelib` to the file /etc/ld.so.conf and then run `ldconfig`. Both of these operations require root privileges. If you do not have root privileges on the system where you encounter this problem, you can follow the advice of the second suggestion in the message and add `/usr/local/somelib` to your `LD_LIBRARY_PATH` environment variable, as in the following example:

```
$ export LD_LIBRARY_PATH=/usr/local/somelib:$LD_LIBRARY_PATH
```

When building applications for your system, you should always check the build logs or output window for messages of the sort shown previously, and make sure that you've added any new library locations to /etc/ld.so.conf on systems where you have root privileges (this is the last suggestion in the bullet list in the message shown earlier in this section). You can test whether adding a specific directory to your library path will resolve shared library problems by manually executing a command such as the following, where `/full/path/to/new/shared/library/dir` is the full path to the directory containing the new shared library, and *ProgramName* is the name of your application:

```
$ /lib/ld.so.version --library-path /full/path/to/new/shared/library/dir ProgramName
```

You can then run this command multiple times, using the `--verify` and `--list` options to ensure that the right directories are being searched and that the right libraries are being found.

You can, of course, eliminate shared library problems in general by linking your application statically. You can do this by adding the `--static` flag to the `CFLAGS` used during compilation (if you are building C applications), but this increases the size of your binary and eliminates both the application size and the library update advantages of using shared libraries.

'No Such File or Directory' Errors

Seeing a "No such file or directory" error when trying to execute a binary that you know is present (such as /bin/ls) can be maddening. This is an interesting twist on the "foo: command not found" error that you get when you try to execute a command that does not actually exist.

Luckily, the cause and solution are usually quite simple—this error indicates a problem with the shared library loader, usually /lib/ld-linux.so.X or /lib/ld.so.X. The most common cause of this problem is a bad symbolic link to the actual loader, which is usually the file /lib/ld-linux.X.Y.Z.so. This error often occurs due to a failed upgrade to a new version of Glibc or incorrectly configured

symlinks in a root filesystem that you created. If the BusyBox binary (busybox) is present on the system where this error occurs, you can usually use BusyBox to resolve the problem as described in the section of Chapter 12 titled "Using BusyBox to Resolve Upgrade Problems." If this error occurs in a root filesystem that you constructed on another system, you can correct the problem using that system's native utilities and then rebuild the root filesystem with the correct links.

Problems Executing Files Compiled with GCC Compilers

After successfully compiling a program using a GCC compiler, the message "cannot execute binary file" is depressing in its simplicity. Luckily, its cause is usually quite simple. This message typically means that the binaries produced by the version of GCC that you executed are actually targeted for a different type of system than the one that you are running on. This usually means that you are running a version of GCC that was configured as a cross-compiler, which runs on one type of system but produces binaries designed to run on another.

Most cross-compilers are built and installed using prefixes that identify the type of system on which they are designed to run. For example, the version of gcc named ppc8xx-linux-gcc is designed to run on an x86 system but produces binaries that are intended to execute on systems with PowerPC 8xx–family processors. However, for convenience' sake, people often create aliases or symbolic links named *gcc* that point to a specific cross-compiler. If you see the message "cannot execute binary file," try the following:

- Use the `alias gcc` command to check that you are not picking up a bad alias or the alias for another compiler. If you see a message like "alias: gcc: not found," this is not the problem.
- Verify that the binary produced by gcc (for example) is actually compiled for the architecture on which you are running by executing the `file filename` command. If its output does not match the type of system on which you are running, you may have accidentally invoked a version of gcc that was built as a cross-compiler for another type of system. You can also execute the compiler with the `--dumpmachine` option (e.g., `gcc –dumpmachine`) to identify the type of system that your version of gcc is producing output for.
- Verify which version of gcc (or other compiler) you are actually running by executing a command such as `which gcc`. In this case, this would return the first version of gcc that is currently found in your path. Make sure that this is the version of gcc that you actually want to run. If it is not, you can adjust the value of your `PATH` environment variable so that the directory containing the version that you want to run is found first, as described in the previous section "Using Multiple Versions of GCC on a Single System."

Running Out of Memory When Using GCC

This problem is uncommon and is usually only seen when using GCC compilers on System V–based systems with broken versions of `malloc()`, such as SCO Unix. In general, I hope that you are not using SCO Unix for anything; but if you are stuck with it for some reason, the easiest solution is to provide a working (i.e., nonsystem) version of `malloc`, the Linux/Unix memory allocation function. The GNU C Library, Glibc, includes a version of `malloc()` that you can build stand-alone. Other popular `malloc()` replacements are the Hoard memory allocator (`http://www.cs.umass.edu/~emery/hoard`), which is especially efficient in multiprocessor environments, and Wolfram Gloger's `ptmalloc()` implementation (`http://www.malloc.de/en`). After building and installing whichever `malloc()` you have chosen, you can relink GCC, specifying the name of your new version of `malloc()` using a command-line argument such as the following:

```
MALLOC=/usr/local/lib/libgmalloc.a
```

■ **Tip** If you have installed and built GCC on your system, you can simply recompile the file gmalloc.c and pass it as an argument when relinking your GCC compilers:

```
MALLOC=gmalloc.o
```

Moving GCC After Installation

To be blunt, **do not attempt to move GCC and associated programs to another directory after building and installing them.** It is always better to rebuild GCC than to try to trick it into running from a directory other than the one for which it was built.

■ **Note** This section generally applies to using GCC on actual Linux systems. If you are using Cygwin, GCC is built so that it can be moved to other directories (i.e., it is relocatable).

As mentioned previously, the GCC C compiler binary (gcc) is primarily a driver program, using spec files to determine how to run the programs associated with each stage of the compilation process (the C preprocessor, linker, loader, and so on). However, gcc itself currently contains hard-coded strings that identify the directory where that version of GCC expects to find these programs. Patches are available for GCC to make it possible to move a prebuilt version of GCC by dumping the spec files, editing them, and then configuring the compiler to use the updated spec files; but this is an accident waiting to happen. Do you really need to do this?

If the gcc binary cannot find one or more of the programs or libraries that it requires, you will see messages such as the following whenever you try to compile a program using GCC.

```
gcc: installation problem, cannot exec 'cc1': No such file or directory
```

This specific message means that the gcc binary was unable to find the program cc1, which is the first stage of the C preprocessor, and is located in gcc's library directory, gcc-lib. If you are truly out of space somewhere, you can move gcc's library directory to another location and then create a symbolic link from the correct location to the new one. For newer versions of GCC, this directory consumes around 35MB of disk space, which is not a huge amount of space by today's standards.

The worst thing about moving something and using symbolic links to glue things back together is that you will almost certainly forget that you have done this at some point. This makes it easy to delete a random directory somewhere, forgetting that symbolic links elsewhere point to it. Unless your backup strategy is a lot more robust than mine, you will almost certainly mess yourself up by doing this. The unfortunate thing about breaking your current GCC installation is that it prevents you from building another version of GCC, forcing you to resort to downloading and installing precompiled binaries in package or gzipped tar file formats.

General Issues in Mixing GNU and Other Toolchains

Most GCC users find that the GCC toolchain is a complete solution for all of their compilation needs. However, there may be times when you need to mix the GCC compilers with components of the compilation food chain that are provided by third parties. This is often the case when using optimized tools from third parties, or when you yourself are developing an enhanced preprocessor or other

part of the compiler toolchain. The following are general suggestions for resolving problems when trying to integrate third-party tools with GCC:

- Check the release notes and documents for both the foreign toolchain and the version of GCC that you are using, in order to identify any known interoperability issues. The next section of this chapter discusses some specific known problems.
- Search the Web for information about known interoperability problems.
- Verify that the problem is actually where you think it is. If you are using a version of the make program other than GNU make, ensure that your version of make supports integrating other tools and passing mandatory command-line options to them correctly. If you are not using GNU make and your version of the make program does not do this correctly, perhaps you should consider performing a free upgrade for your existing make program by downloading and installing GNU make. You can execute the `make –dry-run` command to see a detailed list of what is being passed to the GCC compiler that you are using. You should also make sure that you're executing the version of make that you intended to execute. For example, some Cygwin systems have multiple versions of make installed.
- Verify that the GCC compiler that you are using can actually find the application or library that you are trying to integrate. You can display any GCC compiler's idea of the current search path by using the `--print-search-dirs` option, as in `gcc --print-search-dirs`. GCC may either not be finding a required library or may be finding the wrong one.
- Verify that the GCC compiler that you are using is executing the right subprograms and finding the correct libraries. For example, you could determine what version of the library library.a your compiler is finding by adding `--print-file-name=library.a` to the `CFLAGS` that you're using during compilation, if compiling C applications. As a similar example, you could determine what version of the cc1 subprogram your gcc compiler is executing by adding `--print-prog-name=cc1` to the `CFLAGS` that you are using during compilation.
- If you are replacing portions of the GCC toolchain with commercial or other tools and experience problems, first make sure that you are correctly integrating the two sets of tools. All commercial toolchains support their own sets of command-line options and may invoke subcomponents (such as an assembler or preprocessor) with options that you may not be aware of because they are supplied internally.
- Use verbose modes for both GCC and whatever tools you are integrating with it. You may be able to spot missing command-line options that you can then pass to the appropriate component of the toolchain by using the appropriate version of GCC's `-X` option. GCC compilers provide the `-Xa` option to pass specific arguments to the assembler, the `-Xl` option to pass arguments to the linker, and the `-Xp` option to pass options to the preprocessor. Once you have found a combination that works for you, you can encapsulate these options by adding them to the contents of your Makefile or shell's `CFLAGS` environment variable.
- If you cannot use primary GCC command-line options to "do the right thing" when invoking commercial or other non-GCC toolchain components, you may be able to modify environment variables in your Makefile to add new command-line options to those passed to the appropriate tool during the compilation process. If you are using GNU make, each of the components in the GCC toolchain (preprocessor, assembler, linker) has a corresponding `FLAGS` environment variable (`ASFLAGS`, `LDFLAGS`, `CPPFLAGS`) that can be set in your Makefiles in order to specify command-line options that are always supplied to the appropriate tool. If you are not using GNU make, your version of the make program does not support these flags. If you are using GCC, perhaps you should consider upgrading your existing make program to GNU make.

■ **Tip** If you modify your Makefile, you can execute the `make -n` command to show the commands that would be executed to recompile your application without actually executing them.

- If the version of the make program that you are using does not support modifying the flags that you are passing to each phase of the compilation process, it may still enable you to set Makefile variables for the tools themselves. If so, you can often include specific sets of options in those application definitions. For example, GNU make supports Makefile environment variables that identify the assembler (`AS`) and the C preprocessor (`CXX`). You may be able to set these variables to the name of the binary for your third-party software plus whatever command-line options are necessary to cause your third-party software to "play nice" with GCC.
- If you are still experiencing problems, talk to the vendor. They may not be happy to know that you want to use their expensive software in conjunction with a free tool, but you have paid for support. Depending on where the problem seems to appear, you can use some combination of the GCC compilers' many debugging options (`-X`, `-c`, `-save-temps`, and `-E` come to mind) to preserve temporary files or halt the compilation process at the point at which the problem materializes. For example, if your third-party assembler cannot correctly assemble code generated by gcc, you can halt gcc's compilation process just before entering the assembler, or simply preserve the temporary input files used by the GNU assembler and provide that to your vendor so they can attempt to identify the problem.
- Make sure that the third-party tool that you are using conforms to the same C or C++ standards as the GCC defaults. See Appendix A for a discussion of using the `-std=`*`STD`* command-line option to specify a different standard for GCC to conform to during compilation.
- The GCC's C++ compiler uses a different application binary interface (ABI) than other C++ compilers. Similarly, GCC's C++ compiler does not do name mangling in the same way as other C++ compilers. For these reasons, object files compiled with another C++ compiler cannot be used with object files compiled using G++. You must recompile your entire application using one or the other. Though this may cause you a bit of work, it was done intentionally to protect you from more subtle problems related to the internal details of your C++ compiler's implementation. If G++ used standard name encoding, programs would link against libraries built using or provided with other compilers, but would randomly crash when executed. Using a unique name-encoding mechanism enables incompatible libraries to be detected at link time, rather than at runtime.
- Similarly, make sure you are not using GNU C or C++ extensions that other tools may not recognize.

The preceding is a general list of interoperability problems. The next section provides specific examples of interoperability and integration problems that are discussed to varying degrees in the GCC documentation itself.

Specific Compatibility Problems in Mixing GCC with Other Tools

This section lists various difficulties encountered in using GCC together with other compilers or with the assemblers, linkers, libraries, and debuggers on certain systems. The items discussed in this section were all listed in the documentation provided with various releases of GCC. If you are experiencing problems using GCC on one platform but not another, make sure that you check GCC's Info file for up-to-date information. You can run Info by using the command `info gcc` after installing GCC. Using Info is explained in detail in Appendix C.

The following are some known interoperability problems on specific platforms:

- On AIX systems, when using the IBM assembler, you may occasionally receive errors from the AIX assembler complaining about displacements that are too large. You can usually eliminate these by making your function smaller or by using the GNU Assembler.
- On AIX systems, when using the IBM assembler, you cannot use the dollar sign in identifiers even if you specify the `-fdollars-in-identifiers` option due to limitations in the assembler. You can resolve this by not using this symbol in identifiers, or by using the GNU Assembler.
- On AIX systems, when compiling C++ applications, shared libraries and dynamic linking do not merge global symbols between libraries and applications when you are linking with `libstdc++.a`. A C++ application linked with AIX system libraries must include the `-Wl,-brtl` option on the linker command line.
- On AIX systems, when compiling C++ applications, an application can interpose its own definition of functions for functions invoked by `libstdc++.a` with "runtime linking" enabled. To enable this to work correctly, applications that depend on this feature must be linked with the runtime-linking option discussed in the previous bullet item and must also export the function. The correct set of link options to use in this case is `-Wl,-brtl,-bE:exportfile`.
- On AIX systems, when compiling NLS-enabled (national language support) applications, you may need to set the `LANG` environment variable to `C` or `En_US` in order to get applications compiled with GCC to link with system libraries or assemble correctly.
- AIX systems do not provide weak symbol support. C++ applications on AIX systems must explicitly instantiate templates. Symbols for static members of templates are not generated.
- Older GDB versions sometimes fail to read the output of GCC versions 2 and later. If you have trouble, make sure that you are using an up-to-date version of GDB (version 6.1 or better). You'll be happier about a newer version of GDB in general.
- If you experience problems using older debuggers such as dbx with programs compiled using GCC, switch to GDB. The pain of the GDB learning curve is much less than the pain of random incompatibilities—plus it is a good career investment.
- If you experience problems using profiling on BSD systems, including some versions of Ultrix, the static variable destructors used in G++ may not execute correctly or at all.
- If you are using a system that requires position-independent code (PIC), the GNU Assembler does not support it. To generate PIC code, you must use some other assembler (usually /bin/as on a Unix-like system).
- If you are linking newly compiled code with object files produced by older versions of GCC, you may encounter problems due to different binary formats. You may need to recompile your old object files in order to link successfully.
- The C compilers on some SGI systems automatically expand the `-lgl_s` option into `-lgl_s -lX11_s -lc_s`. This may cause some IRIX programs that happened to build correctly to fail when you first try to recompile them under GCC. GCC does not do this expansion—you must specify all three options explicitly.
- On SPARC systems, GCC aligns all values of type `double` on an 8-byte boundary and expects every `double` to be aligned in the same fashion. The Sun compiler usually aligns `double` values on 8-byte boundaries, with the exception of function arguments of type `double`. The easiest way around this problem without extensive code changes is to recompile your entire application with GCC. The GCC documentation provides sample code to work around this problem, but patching a torn pair of pants does not make them new—plus you have to maintain the patches.

- On Solaris systems, the `malloc` function in the libmalloc.a library may allocate memory that is only aligned along 4-byte boundaries. Unfortunately, GCC on SPARCs assumes that doubles are aligned along 8-byte boundaries, which may cause a fatal signal if doubles are stored in memory allocated by Sun's libmalloc.a library. The only easy solution to this problem is to use `malloc` and related functions from libc.a rather than using the libmalloc.a library.

Problems When Using Optimization

The key problem with optimization is that sometimes it works too well. When using optimization, there will always be a certain amount of disagreement between an optimized executable and your source code. For example, you may find that local variables cannot be traced or examined when you are debugging an optimized application. This is often caused by the fact that GCC may have optimized the variable out of existence.

In general, you rarely need to use optimization when you are still in the debugging phase of application development. If an application works correctly before optimization but no longer works correctly when you specify optimization levels such as `-O2` or `-O3`, you should make sure that you are actually `malloc`ing every data structure that you access through a pointer. Nonoptimized programs may accidentally work if they access memory that is actually associated with other data structures that are not currently being used. By minimizing allocation and eliminating unnecessary or unused structures, optimization may cause "working" programs to fail in this circumstance.

Problems with Include Files or Libraries

As part of the process of building GCC, GCC runs shell scripts that make local copies of system header files that GCC believes exhibit various types of problems. Most target systems have some number of header files that will not work with GCC because they have bugs, are incompatible with ISO C, or are designed to depend on special features provided by other compilers. The best known of the scripts run by GCC is called fixincludes, for fairly obvious reasons.

After making local copies of these files and updating them, GCC uses these updated system header files instead of the generic system header files. If you subsequently install updated versions of your system header files, an existing GCC installation does not know that the original files have changed. GCC will therefore continue to use its own copies of system header files that may no longer be appropriate to the kernel that your system is running or the libraries that the operating system is currently using.

To resolve this sort of problem, you can either regenerate GCC's header files or (to use the really big hammer) reinstall GCC and cause it to regenerate its local copies of the system header files.

- To simply regenerate GCC's header files, run the mkheaders script, which is installed in the directory `install-root/libexec/gcc/target/version/install-tools`.
- To reinstall GCC and cause it to regenerate its header files, change to the directory in which you built GCC and delete the files named stmp-fixinc and stmp-headers, and the entire `include` subdirectory. You can then rerun the `make install` command, and GCC will recreate its local copies of problematic header files. If this process generates errors, you will need to obtain a newer copy of GCC. Your operating system vendor may have made changes that are extensive enough to make them incompatible with the shell scripts provided with your current version of GCC.

■ **Note** On some older operating systems, such as 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. However, this also means that the directory of fixed header files is good only for the machine model on which it was built. For the most part, this is not an issue because only kernel-level programs should care about the differences between different types of machines. If absolutely necessary, you can build separate sets of fixed header files for various types of machines, but you will have to do this manually. Check the Web for information about scripts that various old-time Sun users have created to do this.

Working with existing system libraries is very different from working with text files such as include files. GCC attempts to be a conforming, freestanding compiler suite. However, beyond the library facilities required by GCC internally, the operating system vendor typically supplies the rest of the C library. If a vendor's C library does not conform to the C standards, programs compiled with GCC compilers that use system libraries may display strange warnings, especially if you use an option such as `-Wall` to make your GCC compiler more sensitive. If you see multiple warnings, and existing applications compiled with the vendor's C compiler exhibit strange behavior when compiled with gcc, you should strongly consider using the GNU C library (commonly called Glibc), which is available as a separate package from the GNU Web site (`http://www.gnu.org/software/libc/libc.html`). Glibc provides ISO C, POSIX, BSD, System V, and X/Open compatibility for Linux systems (and for the GNU project's own HURD-based operating system). Chapter 12 of this book can help you get Glibc installed and working on your system. If you do not want to switch to this or are not running Linux or HURD, your only other alternative is to petition your operating system vendor for newer libraries. You have my sympathy in advance.

Mysterious Warning and Error Messages

The GNU compiler produces two kinds of diagnostics: errors and warnings. Each of these has a different purpose:

- Errors report problems that make it impossible to compile your program. GCC compilers report errors with the source filename and the line number where the problem is apparent.
- Warnings report other unusual conditions in your code that may indicate a problem, although compilation can (and will) proceed. Warning messages also report the source filename and the line number, but include the leading text string "warning" to distinguish them from error messages.

Warnings indicate points in your application that you should verify. This may be due to using questionable syntax, obsolete features, or nonstandard features of GNU C or C++. Many warnings are only issued if you specify the "right" one of GCC's `-W` command-line options (or the `-Wall` option, which turns on a popular selection of warnings).

GCC compilers always try to compile your program if this is at all possible. However, in some cases, the C and C++ standards specify that certain extensions are forbidden. Conforming compilers such as gcc or g++ must issue a diagnostic when these extensions are encountered. For example, the gcc compiler's `-pedantic` option causes gcc to issue warnings in such cases. Using the stricter `-pedantic-errors` option converts such diagnostic warnings into errors that will cause compilation to fail at such points. Only those non-ISO constructs that are required to be flagged by a conforming compiler will generate warnings or errors.

To increase the gcc's tolerance for outdated or older features in existing applications, you can always use the `-traditional` option, which attempts to make gcc behave like a traditional C compiler, following the C language syntax described in the book *The C Programming Language, Second Edition*, Brian Kernighan and Dennis Ritchie (Prentice Hall, 1988. ISBN: 0-131-10362-8). The next section discusses differences and subtleties between the gcc and K&R C compilers (Kernighan and Ritchie represent the *K* and *R* in K&R).

Incompatibilities Between GNU C and K&R C

This section discusses some of the more significant incompatibilities between GNU C and K&R C. In real life, the chances of encountering these problems are small, unless you are recompiling a huge legacy application that was written to work with a K&R C compiler, or, if like the author of this book, you still automatically think of ANSI C as "that new thing."

■ **Note** As mentioned in the previous section, using gcc's `-traditional` command-line option causes gcc to masquerade as a traditional Kernighan and Ritchie C compiler. This emulation is necessarily incomplete, but it is as close as possible to the original.

Some of the more significant incompatibilities between GNU C (the default standard used by gcc) and K&R (non-ISO) versions of C are the following:

- The gcc compiler normally makes string constants read-only. If several identical string constants are used at various points in an application, gcc only stores one copy of the string. One consequence of this is that, by default, you cannot call the `mktemp()` function with an argument that is a string constant, because `mktemp()` always modifies the string that its argument points to. Another consequence is that functions such as `fscanf()`, `scanf()`, and `sscanf()` cannot be used with string constants as their format strings, because these functions also attempt to write to the format string. The best solution to these situations is to change the program to use arrays of character variables that are then initialized from string constants. To be kind, gcc provides the `-fwritable-strings` option, which causes gcc to handle string constants in the traditional, writable fashion. This option is automatically activated if you supply the `-traditional` option.
- To gcc, the value `-2147483648` is positive because `2147483648` cannot fit in an `int` data type. This value is therefore stored in an unsigned `long int`, as per the ISO C rules.
- The gcc compiler does not substitute macro arguments when they appear inside of string constants. For example, the following macro in gcc:

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

  will produce the actual string `"a"` regardless of the value of `a`. Using the `-traditional` option causes gcc to do macro argument substitution in the traditional (old-fashioned) non-ISO way.
- When you use the `setjmp()` and `longjmp()` functions, the only automatic variables guaranteed to remain valid are those declared as volatile. This is a consequence of automatic register allocation. If you use the `-W` option with the `-O` option, gcc will display a warning when it thinks you may be depending on nonvolatile data in conjunction with the use of these functions. Specifying gcc's `-traditional` option causes gcc to put variables on the stack in functions that call `setjmp()`, rather than in registers. This results in the behavior found in traditional C compilers.

- Programs that use preprocessing directives in the middle of macro arguments do not work with gcc. ISO C does not permit such a construct and is poor form anyway. The `-traditional` option will not help you here.
- Declarations of external variables and functions within a block apply only to the block containing the declaration—they have the same scope as any other declaration in the same place. Using the `-traditional` option causes gcc to treat all external declarations as globals, as in traditional C compilers.
- In traditional C, you can combine type modifiers with preexisting `typedef`ed names, as in the following example:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ISO C, this is not allowed. The `-traditional` option cannot change this type of gcc behavior because this grammar rule is expressed in high-level language (Bison, in this case) rather than in C code.
- The gcc compiler treats all characters of an identifier as significant, rather than only the first eight characters used by K&R C. This is true even when the `-traditional` option is used.
- The gcc compiler does not allow whitespace in the middle of compound assignment operators such as +=.
- The gcc compiler complains about unterminated character constants inside of a preprocessing conditional that fails if that conditional contains an English comment that uses an apostrophe. The `-traditional` option suppresses these error messages, though simply enclosing the comment inside comment delimiters will cause the error to disappear. The following is an example:

  ```
  #if 0
      You cannot expect this to work.
  #endif
  ```

- Many older C programs contain declarations such as `long time()`, which was fine when system header files did not declare this function. However, systems with ISO C headers declare `time()` to return `time_t`. If this is not the same size as `long`, you will receive an error message. To solve this problem, include an appropriate system header file (<time.h>) and either remove the local definitions of `time()` or change them to use `time_t` as the return type of the `time()` function.
- When compiling functions that return structures or unions, gcc's output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with gcc cannot call a structure-returning function compiled with an older compiler such as PCC (Portable C Compiler), and vice versa. You can tell gcc to use a compatible convention for all functions that return structures or unions by specifying the `-fpcc-struct-return` option.

Abuse of the __STDC__ Definition

Though more a stylistic or conceptual problem than a gcc issue, the meaning of `__STDC__` is abused frequently enough that it deserves a short discussion here.

Programmers normally use conditionals on whether `__STDC__` is defined in order to determine whether it is safe to use features of ISO C such as function prototypes or ISO token concatenation. Because vanilla gcc supports all the features of ISO C, all such conditionals should test as true, even when the `-ansi` option is not specified. Therefore, gcc currently defines `__STDC__` as long as you do not specify the `-traditional` option. The gcc compiler defines `__STRICT_ANSI__` if you specify the `-ansi` option or an `-std` option specifying strict conformance to some version of ISO C.

Unfortunately, many people seem to assume that `__STDC__` can also be used to check for the availability of certain library facilities. This is actually incorrect in an ISO C program, because the ISO C standard says that a conforming freestanding language implementation should define `__STDC__`, even though it does not have the library facilities. The command `gcc -ansi -pedantic` is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ISO C library.

■ **Tip** You should not automatically undefine `__STDC__` in a C++ application. Many header files are written to provide prototypes in ISO C but not in traditional C. Most of these header files can be used in C++ applications without any changes if `__STDC__` is defined. If `__STDC__` is undefined in a C++ application, they will all fail and will need to be changed to contain explicit tests for C++.

Resolving Build and Installation Problems

Having been successfully installed and used on a huge number of platforms for the past 15 years or so, GCC's build and installation processes are quite robust. GCC requires a significant amount of disk space during the build and test process, especially if you build and test all of the compilers (C, C++, Ada, Java, Fortran) that are currently included in the GNU Compiler Collection. One of the most common problems encountered when building and installing the GCC compilers is simply running out of disk space.

Before you begin to build and install GCC compilers, think about the tools that you are using to build them. For example, it is easier and safer to build GCC using GNU make rather than any system-specific version of make. Similarly, GCC is best built with GCC. Though this seems like a paradox, it really is not. When bootstrapping GCC on Solaris boxes without the official (paid) Solaris C compiler, I usually download a prebuilt binary of the gcc C compiler for Solaris (from a site such as `http://www.sunfreeware.com`), download the latest GCC source code, and then use the former to build the latter. Maybe I'm just paranoid.

■ **Note** Using a compiler to compile itself is an impressive feat, though conceptually an equally impressive exercise in recursion.

If you have previously built and installed GCC and then encounter problems when rebuilding it later, make sure that you did not configure GCC in your primary directory unless you intend to always build it there. The GCC build and installation documents suggest that you create a separate build directory and then execute the configure script from that directory as `../gcc-VERSION/configure`. This will fail if you have previously configured GCC in the primary directory.

As mentioned earlier, a common error encountered when building GCC on NFS-dependent systems is that the fixincludes script that generates GCC local versions of system header files may not work correctly if the directory containing your system header files is automounted—it may be unmounted while the fixincludes script is running. The easiest way to work around this problem is to make a local copy of `/usr/include` under another name, temporarily modify your automounter maps (configuration files) not to automount `/usr/include`, and then restart autofs (on a Linux system) or whatever other automounter daemon and control scripts you are using. If your entire `/usr` directory structure is automounted, such as on many Solaris systems, you should just put together a scratch machine with a local `/usr` directory and build your GCC compilers there. After installing it, archive the installation directory and install your GCC compilers onto other systems from the archive.

Building and installing GCC has been done by thousands of users everywhere. I have built and used GCC on everything from Apollo workstations (back in the day) and a huge variety of embedded systems, all the way up to the 64-bit home Linux boxes that I use today. If you encounter a problem, check the release notes, the README file, and the installation documentation. If you still have problems, feel free to send me e-mail at the address given in the introduction to this book, or visit `http://gcc.gnu.org`, or check the news archives at `http://www.google.com`. If you have experienced a problem, chances are that someone else has too.