The GCC Compilers

Preface

If all you really want to know is how to compile your C or C++ program using GCC, and you don't have the time or interest in understanding what you're doing or what GCC is, you can skip most of these notes and cut to the chase by jumping to the examples in Section 6. I think you will be better off if you take the time to read the whole thing, since I believe that when you understand what something is, you are better able to figure out *how* to use it.

If you have never used GCC, or if you have used it without really knowing what you did, (because you were pretty much using it by rote), then you should read this. If you think you *do* understand GCC and do not use it by rote, you may still benefit from reading this; you might learn something anyway. Because I believe in the importance of historical context, I begin with a brief history of GCC.

1 Brief History

Richard Stallman started the GNU Project in 1984 with the purpose of creating a free, Unix-like operating system. His motivation was to promote freedom and cooperation among users and programmers. Since Unix requires a C compiler and there were no free C compilers at the time, the GNU Project had to build a C compiler from the ground up. The Free Software Foundation was a non-profit organization created to support the work of the GNU Project.

GCC was first released in 1987. This was a significant breakthrough, being the first portable ANSI C optimizing compiler released as free software. Since that time GCC has become one of the most important tools in the development of free software.

In 1992, it was revised and released as GCC 2.0, with the added feature of a C++ compiler. It was revised again in 1997, with improved optimization and C++ support. These features became widely available in the 3.0 release of GCC in 2001.

2 Languages Supported by GCC

GCC stands for "GNU Compiler Collection". GCC is an integrated collection of compilers for several major programming languages, which as of this writing are C, C++, Objective-C, Java, FORTRAN, and Ada. The GNU compilers all generate machine code, not higher-level language code which is then translated via another compiler.

3 Language Standards Supported by GCC

For the most part, GCC supports the major standards and provides the ability to turn them on or off. GCC itself provides features for languages like C and C++ that deviate from certain standards, but turning on the appropriate compiler options will make it check programs against the standards. The section below entitled Consult the manual for details.

4 A Bit About the Compile and Link Process

Suppose that you have written a very simple C program such as the following

```
1:  #include <stdio.h>
2:
3:  int main()
4:  {
5:      printf( "Hello world\n" );
6:      return 0;
7:  }
```

and placed it in a file named `helloworld.c`. You may have been told that you need that first line, which is the `#include` directive, `#include <stdio.h>`, but you may not really know why it is there. The reason is that the `printf()` function is declared in the header file `stdio.h` and in order for the compiler to check that you are calling it correctly in line 5, it needs to compare the declaration of the `printf()` function with its use. The compiler just needs to check things such as the number of parameters, their types, and the return value of the function.

The way this is done is by copying the entire header file into the program before the compiler runs. The `#include` directive literally copies the entire file, `stdio.h`, into your program starting at line 1. This is done by a program called the *C preprocessor*. Once the header file is physically part of your program, the compiler will run and will be able to validate the call to `printf()` by comparing it to the declaration that it read in an earlier line in the modified file.

The header file does not contain the definition of the `printf()` function, i.e., its implementation. That is contained in the C Standard I/O Library. The compiler is not able to create the machine instructions that will cause the `printf()` function to run, because it does not know "where" the `printf()` implementation is; it cannot create a "call" to this function. Instead, the compiler places a notation in the executable file that says, more or less, "the call to `printf()` must be resolved by the linker."

The linker is a separate program that runs after the compiler. It looks at all of the unresolved symbols in the program, such as `printf()`, and tries to resolve them by looking up their locations in the software libraries that the program needs. In this case, the linker needs to look up in the C Standard I/O Library the location of the `printf()` function, so that it can patch the code to make a call to it. The C Standard I/O Library is special because it is used in almost every C program, and therefore many C implementations include it within the C runtime library. This makes it possible for the linker to find it easily.

The same discussion would apply if you wrote the above program in C++ as in

```
1:  #include <iostream>
2:
3:  int main()
4:  {
5:      std::cout << "Hello world\n";
6:      return 0;
7:  }
```

only instead of using the `stdio.h` header file, it would use the `iostream` header file, and the iostream library in C++ instead of the C Standard I/O Library.

In summary, a header file contains declarations that the compiler needs, but not implementations. The corresponding library file has those. The compiler needs the header files but not the libraries; the linker needs the libraries, not the header files.

5 Command Options and Control

From this point forward, lowercase *gcc* will refer to the executable program name, i.e., what you type to run the GCC compiler.

5.1 Options and File Extensions Controlling the Kind of Output

When you run *gcc*, it usually performs preprocessing, compiling, assembly and linking. There are options to control which of these steps are performed. *gcc* can also look at the file extension for guidance as to which compiler to use and what kind of output to generate. For example, a file ending in `.c` is assumed to be C source code, and files ending in either `.cc`, `.cpp`, `.c++`, `.C` , and `.cxx` are taken to be C++ source code. (There are other extensions that are also taken to imply C++ source code.) You should consult the manual for other extensions and languages. The following options are either very useful or enlightening.

`-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`'. E.g.,

```
gcc -c myprog.c
```

produces `myprog.o`.

`-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`'. It is unlikely that you will need to do this, but it is educational to look at the output of the compiler, which is in assembly language.

`-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. E.g.,

```
gcc -E myprog.c > myprog.i
```

This is also an educational exercise – you can see for yourself what the preprocessor does to your source code, to get a better understanding of how to use it.

`-o` ***`file`*** Place output in file ***`file`***. This applies regardless of whatever sort of output is being produced, whether it is an executable file, an object file, an assembler file or preprocessed C code. Usually you use this to name your executable. E.g.,

```
gcc -o myprog myprog.c
```

`-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.

`--help` Print (on the standard output) a description of the command line options understood by *gcc*.

5.2 Compiling C++ Programs

gcc comes with a compiler named *g++* that specifically compiles C++ programs, regardless of the file extension. Sometimes you need to use the C++ compiler even though the file extension is not a C++ extension; in this case you need to use *g++*.

5.3 Options that Control the C Dialect

By "dialect" is meant a specific collection of features of C. For example, the ANSI standard known as ISO90 C is a dialect of C. The full set of features supported by *gcc* is much larger than the ANSI standard, and this is also a dialect. Another dialect is obtained by adding GNU extensions to the ANSI ISO C90 standard. You can selectively remove features from the full GNU set of extensions. The basic options, however, are:

`-ansi` In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions that conflict with ISO C++. This turns off certain features of *gcc* that are incompatible with ISO C90 (when compiling C code), or of standard C++ (when compiling C++ code).
For example, the getline() function is a GNU extension to C. It is not in ANSI C. If your program myprog.c contains its own getline() function then if you compile with the line

```
gcc myprog.c
```

you will get the error

```
temp4.c:4: error: conflicting types for 'getline'
```

but if you use

```
gcc -ansi myprog.c
```

the GNU extensions will be disabled, and there will be no type conflict.

`-std=` When followed by a specific dialect designating string such as '`c90`' or '`gnu9x`', it specifies that dialect.

`-fno-...` There are many options that begin with `-fno-` and are followed by a string that represents some feature to disable. The '`no`' means 'turn off'. Consult the manual.

5.4 Options that Control Warnings

Warnings are diagnostic messages about constructions that are not errors, but are often associated with errors. For example, when a variable is declared but never used in a program, it is possible that the programmer overlooked something, so the compiler could issue a warning when it finds such a construction. gcc allows you to suppress certain warnings and request others. Some of the more common options are listed below.

`-w`	Inhibit all warning messages.
`-Wall`	Enable all warning messages (recommended for all development work).
`-Wextra`	Enable warnings that are not checked by `-Wall`. For example, comparing an `unsigned int` variable against `-1` suggests that you forgot that the variable was unsigned.

5.5 Options For Debugging

`-g` This produces debugging information in the operating system's native format. 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. E.g.,

```
gcc -g -o myprog myprog.c
```

Once you have compiled a program with debugging information, you can run it under the control of the debugger. Of course you have to learn how to use the debugger, *gdb*, from the command line, or from within a GUI-based SDK that uses it as the underlying debugger.

5.6 Options Controlling the Preprocessor

Of course you are aware of the fact that when your program is compiled, the very first step that the compiler takes is to run the preprocessor, which processes all of the preprocessor directives, those lines that begin with the pound sign '#'. There are options that control how the preprocessor behaves, and these are important to know and understand. The most important are (1) how to define symbols on the command line, and (2) how to tell the preprocessor where to look for include-files.

`-D` *`name`* This predefines *`name`* as a macro symbol, with the value 1, or true if you want to think of it that way.

`-D` *`name=definition`* The contents of definition are tokenized and processed as if they appeared during translation phase three in a '`#define`' directive. E.g.,

```
gcc -D testvalue=6 -o myprog myprog.c
```

is the same as if you had placed

```
#define testvalue 6
```

in `myprog.c`

If you are invoking the preprocessor from a shell or shell-like program you need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax. For example,

```
gcc -D 'introduction=AArgh$#^*!!' -o myprog myprog.c
```

would protect the meta-characters in `AArgh$#^*!!` from the shell, so that introduction contains that string exactly.

`-U` *`name`*	Cancel any previous definition of name, either built in or provided with a '`-D`' option.
`-undef`	Do not predefine any system-specific or gcc-specific macros. The standard predefined macros remain defined.

Any definitions or undefinitions in the program files override definitions or undefinitions you make on the command line. Thus, if you define a symbol on the command line but within the program you undef it, it will be undef-ed from that point forward.

5.7 Options for Linking

When *gcc* finds unresolved symbols in your program, it has to resolve them by searching in library files. You specify nonstandard libraries using the following option.

`-l`*`library`*
`-l` *`library`*

Search the library named `library` when linking. (The second alternative with the `library` as a separate argument is only for POSIX compliance and is not recommended.) It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the order they are specified.
IMPORTANT: If your program references the symbol `supersort` and that symbol is defined in the library `libgreatstuff.a`, then your command line would have to be

```
gcc -o myprog myprog.c -lgreatstuff
```

because *gcc* does not know what it has to look for until it reads `myprog.c`'s unresolved symbol list, and so it is only after seeing `myprog.c` that it will search for the symbol `supersort`. If you reverse the two words on the command line, you will get a linker error.
Note that the name you supply to -l is not the full library name, but the name with the 'lib' and the '.a' removed.

5.8 Options to Control Directory Search

`-I` *`dir`*

Add the directory *`dir`* to the list of directories to be searched for *header files*. Directories named by '`-I`' are searched before the standard system include-directories. If the directory *`dir`* is a standard system include directory, the option is ignored to ensure that the default search order for system directories and the special treatment of system headers are not defeated.
It does not matter whether there is space between the `I` and the directory name.

The `-I` option is important. It is the way to tell *gcc* that an included header file is not in any of the standard places in which it looks. For example, suppose that your program needs the header file `utilities.h`, which is in the directory `~/my_includes`. The program contains the line

```
#include “utilities.h”
```

Your command line must contain "`-I ~/my_includes`" otherwise *gcc* will report that it cannot find the file `utilities.h`.

The `-I` option controls where *gcc* looks for header files. It does not tell *gcc* where to look for libraries. When *gcc* links your program, it has to find all library modules that your program uses. If you are using libraries that are not in *gcc*'s library search path, then you must tell it on the command line to search the containing directories. The `-L` option does this:

`-L dir`	Add `dir` to *gcc*'s search path for libraries.

For example, suppose that the program `myprog.c` makes a call to a function named `log_error()` that is declared in the `my_utilities.h` header file and is defined in the library `libutilities.a` located in the directory `~/my_libs`. You could write

```
gcc -o myprog myprog.c -I ~/my_includes -lutilities -L ~/my_libs
```

so that *gcc* looks in the `my_libs` directory for the file. Alternatively, you could write

```
gcc -o myprog myprog.c -I ~/my_includes -l ~/my_libs/libutilities.a
```

which is equivalent.

5.9 Environment Variables Affecting GCC

Certain environment variables change the way *gcc* behaves. Initially the ones of some importance are the variables that tell it where to search for included files, where to search for libraries, and where to search for dynamically loaded libraries. If you are writing programs that are internationalized and you need to make sure that locale information is specified, you will also need to set some environment variables. The variables of interest are therefore

`C_INCLUDE_PATH`	A colon-separated list of directories in which to look for include files for C programs.
`CPLUS_INCLUDE_PATH`	A colon-separated list of directories in which to look for include files for C++ programs.
`LIBRARY_PATH`	A colon-separated list of directories that the linker uses to look for static libraries.
`LD_LIBRARY_PATH`	A colon-separated list of directories that the linking loader uses to look for dynamic libraries.
`LANG`	A variable that controls the locale information used when the compiler is parsing strings and comments in the program.

6 Examples

Here is a collection of examples to demonstrate how to do the typical tasks. If you did not read about the supported languages in Section 2 above, then make a note to yourself that GCC will accept C++ programs with any of the extensions `.cc`, `.cpp`, `.cxx`, `.c++`, and `.C`. I will use `.cpp` extensions to denote C++ programs. In all cases, I have used the `-Wall` option to enable all warnings. If you learn a bit about using the GDB debugger, then you would be wise to include the `-g` option as well.

Single Source File Programs

1. To compile a program entirely contained in a single file named `fudge.c`, putting the executable code into a file named `fudge`:

   ```
   gcc -Wall -o fudge fudge.c
   ```

2. To compile the C++ program `fudge.cpp` putting the executable code into `fudge`:

   ```
   g++ -Wall -o fudge fudge.cpp
   ```

3. If you are really in a hurry to compile `fudge.cpp` and you don't even have the time to give the executable a name, just type:

   ```
   g++ fudge.cpp
   ```

   In this case, *g++* will place the executable in a file named `a.out`, overwriting any other `a.out` that existed in your current working directory. Other than sheer laziness, the only reason to do this is to check quickly if the program compiles.

Multi-Source File Programs

1. If you have a program distributed among the files `nuts.c`, `fudge.c`, and `ice_cream.c`, each of which has a corresponding header file, `nuts.h`, `fudge.h`, and `ice_cream.h`, all of which are included in the main program, `sundae.c`, then enter the following command to create the `sundae` executable:

   ```
   gcc -Wall -o sundae nuts.c fudge.c ice_cream.c sundae.c
   ```

   Whereas *you* may care whether the nuts precede the fudge or the ice cream in your sundae, *gcc*'s linker does not care. *gcc* will be just as happy if you write

   ```
   gcc -o sundae sundae.c fudge.c ice_cream.c nuts.c
   ```

   or any other rearrangement of the source code file names.

2. Suppose that you modified the `nuts.c` file (maybe you're using walnuts instead of pecans now). If you type the entire line written above, you will be recompiling every other file needlessly. It is much faster and more efficient to compile each of the source code files separately. This is a multi-step procedure:

   ```
   gcc -c nuts.c fudge.c ice_cream.c sundae.c
   gcc -o sundae nuts.o fudge.o ice_cream.o sundae.o
   ```

   The first line produces object files, i.e. .o files, for each .c file. Thus, *gcc* will create `nuts.o`, `fudge.o`, `ice_cream.o`, and `sundae.o`. The second line implicitly invokes the linker to link all of these object files together and produce the executable, `sundae`. It does not matter in which order you write the object files on the line.

   The linker also links any libraries that the program requires into the executable (provided that it can find them; please read Section 5.7, Options for Linking, and Section 5.8, Options to Control Directory Search, to better understand.) Now, if you change `nuts.c` all you have to do is (1) recompile `nuts.c`:

   ```
   gcc -c nuts.c
   ```

   and (2) re-link it using the second line:

   ```
   gcc -o sundae nuts.o fudge.o ice_cream.o sundae.o
   ```

   Linking is usually faster than compiling, so this will save you time.

3. If you do this often enough you will discover that it is a drag to have to keep re-typing the same commands over and over. This is why the make program and Makefiles were invented. But that is another lesson.