Compiling C and C++ Programs

gcc is the "GNU" C Compiler, and **g++** is the "GNU C++ compiler, while **cc** and **CC** are the Sun C and C++ compilers also available on Sun workstations. Below are several examples that show how to use g++ to compile C++ programs, although much of the information applies to C programs as well as compiling with the other compilers.

Example 1: Compiling a simple program

Consider the following example: Let "hello.C" be a file that contains the following C++ code.

```
#include "iostream.h"
int main()
{
  cout << "Hello\n";
}
```

The standard way to compile this program is with the command

```
g++ hello.C -o hello
```

This command compiles hello.C into an executable program named "hello" that you run by typing 'hello' at the command line. It does nothing more than print the word "hello" on the screen.

Alternatively, the above program could be compiled using the following two commands.

```
g++ -c hello.C
g++ hello.o -o hello
```

The end result is the same, but this two-step method first *compiles* hello.C into a machine code file named "hello.o" and then *links* hello.o with some system libraries to produce the final program "hello". In fact the first method also does this two-stage process of compiling and linking, but the stages are done transparently, and the intermediate file "hello.o" is deleted in the process.

Frequently used compilation options

C and C++ compilers allow for many options for how to compile a program, and the examples below demonstrate how to use many of the more commonly used options. In each example, "myprog.C" contains C++ source code for the executable "myprog". In most cases options can be combined, although it is generally not useful to use "debugging" and "optimization" options together.

Compile myprog.C so that myprog contains symbolic information that enables it to be debugged with the **gdb** debugger.

```
g++ -g myprog.C -o myprog
```

Have the compiler generate many warnings about syntactically correct but questionable looking code. It is good practice to *always* use this option with gcc and g++.

```
g++ -Wall myprog.C -o myprog
```

Generate symbolic information for gdb *and* many warning messages.

```
g++ -g -Wall myprog.C -o myprog
```

Generate optimized code on a Solaris machine with warnings. The -O is a capital o and not the number 0!

```
g++ -Wall -O -mv8 myprog.C -o myprog
```

Generate optimized code on a Solaris machine using Sun's own CC compiler. This code will generally be faster than g++ optimized code.

```
CC -fast myprog.C -o myprog
```

Generate optimized code on a Linux machine.

```
g++ -O myprog.C -o myprog
```

Compile myprog.C when it contains Xlib graphics routines.

```
g++ myprog.C -o myprog -lX11
```

If "myprog.c" is a C program, then the above commands will all work by replacing g++ with gcc and "myprog.C" with "myprog.c". Below are a few examples that apply only to C programs.

Compile a C program that uses math functions such as "sqrt".

```
gcc myprog.C -o myprog -lm
```

Compile a C program with the "electric fence" library. This library, available on all the Linux machines, causes many incorrectly written programs to crash as soon as an error occurs. It is useful for debugging as the error location can be quickly determined using gdb. However, it should only be used for debugging as the executable myprog will be much slower and use much more memory than usual.

```
gcc -g myprog.C -o myprog -lefence
```

Example 2: Compiling a program with multiple source files

If the source code is in several files, say "file1.C" and "file2.C", then they can be compiled into an executable program named "myprog" using the following command:

```
g++ file1.C file2.C -o myprog
```

The same result can be achieved using the following three commands:

```
g++ -c file1.C
g++ -c file2.C
g++ file1.o file2.o -o myprog
```

The advantage of the second method is that it compiles each of the source files separately. If, for instance, the above commands were used to create "myprog", and "file1.C" was subsequently modified, then the following commands would correctly update "myprog".

```
g++ -c file1.C
g++ file1.o file2.o -o myprog
```

Note that file2.C does not need to be recompiled, so the time required to rebuild myprog is shorter than if the first method for compiling myprog were used. When there are numerous source file, and a change is only made to one of them, the time savings can be significant. This process, though somewhat complicated, is generally handled automatically by a makefile.