gcc/g++ Tutorial

Dr. Yihsiang Liow (September 17, 2022)

Contents

1 Introduction

The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

Originally named the GNU C Compiler, because it only handled the C programming language, GCC 1.0 was released in 1987 and the compiler was extended to compile C++ in December of that year. Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.

As well as being the official compiler of the unfinished GNU operating system, GCC has been adopted as the standard compiler by most other modern Unix-like computer operating systems, including Linux and the BSD family. A port to RISC OS has also been developed extensively in recent years. There is also an old (3.0) port of GCC to Plan9, running under its ANSI/POSIX Environment (APE). GCC is also available for Microsoft Windows operating systems and for the ARM processor used by many portable devices.

GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in proprietary development environments. GCC is also available for most embedded platforms, including Symbian (called gcce), AMCC, and Freescale Power Architecture-based chips. The compiler can target a wide variety of platforms, including videogame consoles such as the PlayStation 2 and Dreamcast. Several companies make a business out of supplying and supporting GCC ports to various platforms, and chip manufacturers today consider a GCC port almost essential to the success of an architecture.

...

The standard compiler releases since 4.6 include front ends for C (gcc), C++ (g++), Objective–C, Objective–C++, Fortran (gfortran), Java (gcj), Ada (GNAT), and Go (gccgo). Also available, but not in standard are Pascal (gpc), Mercury, Modula-2, Modula-3, PL/I, D (gdc), and VHDL (ghdl). A popular parallel language extension, OpenMP, is also supported.

The Fortran front end was g77 before version 4.0, which only supports FORTRAN 77. In newer versions, g77 is dropped in favor of the new gfortran front end that supports Fortran 95 and parts of Fortran 2003 as well.[36] As the later Fortran standards incorporate the F77 standard, standards-compliant F77 code is also standards-compliant F90/95 code, and so can be compiled without trouble in gfortran. A front-end for CHILL was dropped due to a lack of maintenance.[37]

A few experimental branches exist to support additional languages, such as the GCC UPC compiler for Unified Parallel C.

– Wikipedia

2 Prerequisites

I assume the following:

- You have a copy of my Fedora virtual machine and have gone over my tutorial on using VMware Player/Workstation and the Fedora virtual machine.
- You have already read *Unix Tutorial 1*.

The C/C++ compiler used is `g++`. If you're using my virtual machine the software is already installed. Otherwise you can install it by doing this:

```
dnf -y install gcc-g++
```

as root.

3 Compiling a Single C/C++ Source File into an Executable

Go ahead and create a directory for all examples in this set of notes. I'm using a directory called `test`.

Create a simple `helloworld.cpp` in your test directory:

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

To compile your C++ source program into an executable, you can do this:

```
[student@localhost test] g++ helloworld.cpp
```

This will build an executable called `a.out`:

```
[student@localhost test] ls -la
total 39
drwxrwxrwx. 1 root root     0 Sep 17 17:08 .
drwxrwxrwx. 1 root root 16384 Sep 17 17:08 ..
-rwxrwxrwx. 1 root root 22640 Sep 17 17:08 a.out
-rwxrwxrwx. 1 root root    94 Sep 17 17:08 helloworld.cpp
```

To run the executable you do this:

```
[student@localhost test] ./a.out
hello world
```

The . means "the current working directory". So you're trying to run the `a.out` in *this* directory. In general to run an executable that is in your current working directory you type this at the prompt:

```
./[executable filename]
```

Of course to run a program `hackgoogle` in directory `/home/jripper/secret/`, you do this:

```
[student@localhost test] /home/jripper/secret/hackgoogle
```

Suppose you want to call your executable `helloworld` instead of `a.out`. You do this:

```
[student@localhost test] g++ helloworld.cpp -o helloworld
```

The `-o` is the output option. You're telling `g++` to send the output (the executable) to the file `helloworld`.

At this point I have

```
[student@localhost test] ls -la
total 62
drwxrwxrwx. 1 root root     0 Sep 17 17:08 .
drwxrwxrwx. 1 root root 16384 Sep 17 17:08 ..
-rwxrwxrwx. 1 root root 22640 Sep 17 17:08 a.out
-rwxrwxrwx. 1 root root 22640 Sep 17 17:08 helloworld
-rwxrwxrwx. 1 root root    94 Sep 17 17:08 helloworld.cpp
```

And of course you run `helloworld` by doing this:

```
[student@localhost test] ./helloworld
hello world
```

If you're also using a GUI, the GUI might display a special icon to tell you that a certain file is an executable. GUIs usually determine the nature of a file by the file extension. For an executable, the default extension is `.exe`. So you might want to name your executable `helloworld.exe` instead:

```
[student@localhost test] g++ helloworld.cpp -o helloworld.exe
```

Exercise 3.1. If your source file is `predictstockprice.cpp` and the executable you want to build is `secret`, what is the command to execute in the shell? What do you type at the shell to run `secret`? ☐

Exercise 3.2. For practice, take any C/C++ book and do 10 programs, building the executable using `g++`. ☐

3.1 Note for C programmers

For a C program, everything above is the same except that you can use `gcc` instead of `g++`. Note that since C is a subset of C++, you can also compile C programs using `g++`.

Exercise 3.3. Compile the following C program `helloworld.c`:

```
#include <stdio.h>

int main()
{
    printf("hello world\n");
    return 0;
}
```

into an executable. Run it. Depending your compiler, the first line of the program might be

```
#include <cstdio>
```

or

```
#include "stdio.h"
```

instead. ☐

Exercise 3.4. Can you compile a C++ program using `gcc`? ☐

Exercise 3.5. Can you compile a C program using `g++`? ☐

3.2 Note to Cygwin users

Executing “`g++ helloworld.cpp`” will produce `a.exe` and not `a.out`. To run `a.exe` you can either execute “`./a`” or “`./a.exe`”.

4 The Warn all Option

There's a difference between a warning and an error. An error reported by g++ means the program is not compiled. A warning is a warning to you that maybe you want to look at your code, but g++ can still compile the program.

Change your `helloworld.cpp` to the following:

```
#include <iostream>

int main()
{
    int x;
    std::cout << "hello world" << std::endl;
    return 0;
}
```

When you compile

```
[student@localhost test] g++ helloworld.cpp
```

gpp has no quarrels with you. Now try this:

```
[student@localhost test] g++ -Wall helloworld.cpp
```

You'll get this message from g++:

```
main.cpp: In function ‘int main()’:
main.cpp:5:9: warning: unused variable ‘x’ [-Wunused-variable]
    5 |     int x;
      |
```

But if you check, you'll see that g++ has given you your `a.out`.

In the above, the `Wall` is called the "warn all" option. This will make g++ print all warnings. Obviously this is a good idea. You are strongly advised to turn on the warn all option. Sometimes a compiler can fix some problems for you. Sometimes a compiler relaxes the actual requirement for the language. But this can be dangerous. The compiler can make wrong decisions. The best is to get the compiler to reveal to you all the warnings during compilation and you decide how to fix the warnings.

By default g++ might or might not print warnings (depending on the type of warning). Try the following for yourself:

```
#include <iostream>

int main()
{
    int x;
    std::cout << x << '\n'; // using x without giving x a value
    return 0;
}
```

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    // No int value returned
}
```

```
#include <iostream>

int f()
{} // No int value returned

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

```
#include <iostream>

int f()
{
    int x = 0;
    if (x == 0)
    {
         return 42;
    }
    // No int return value here
}

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Altogether with the output option, you can do this:

```
[student@localhost test] g++ -Wall helloworld.cpp -o helloworld.exe
```

Recall from the above that even when you have warnings g++ will still produce an executable. Obviously this is dangerous. What you work at a nuclear power plant, compiled your C++ program, had some warnings, went off to get coffee, came back and forgot about the warnings, and ran your executable?

One flag that will help is the `Werror` option. This will convert all warnings to errors, which means that g++ will treat warnings as errors and therefore if there are warnings, then an executable will not be produced. (Of course you might previously have an executable in that directory.)

Another option that you might want to use is `Wextra` that will provide extra warnings. This will for instance trigger a warning if you have an unsigned int `x` and your code compares "`x >= 0`", which is curious since all unsigned int values are non-negative anyway.

There's also the `pedantic` option that checks if your code is ANSI standard compliant. This option is probably not necessary for you.

So if you want to include all the above options, you would compile like this:

```
g++ -Wall -Wextra -Werror -pedantic main.cpp -o main.exe
```

(Yes personally I do use these options.)

5 Overview of Compilation Process

Given a C++ program like this

```
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

the process that produces an executable is made up of several steps.

1. Preprocessing: The preprocessor directives, i.e., lines that start with # such as

   ```
   #include <iostream>
   ```

   are processed. For `#include <iostream>`, this line is replaced by the contents of the file `iostream.h`.

2. Compilation: The compiler reads each source file and build an object file.

3. Link: All the relevant object files are linked to form to produce an executable file.

(See CISS245 notes.) Of course the above is just a quick overview – the real picture is a lot more complicated. For instance the compilation step can be broken down into:

2.1. Statements are read and assembly code is produced

2.2. The assembly code is converted to machine code

There are usually some optimization steps as well.

6 Preprocessing Option

The following is useful when you think something is wrong with the preprocessing step (#include, etc.) If you compile with the preprocessing option, only the preprocessing step is executed.

First write two files: t.cpp and s.h. t.cpp will #include the file s.h. Using these two files, I'll show you the preprocessor in action.

First here is t.cpp:

```
#include "s.h"

int main()
{
    return 0;
}
```

Here's the second file s.h:

```
#ifndef S_H
#define S_H

void f();

#endif
```

Now if you run

```
[student@localhost test] gcc t.cpp -E -o t1.cpp
```

This will send the preprocessed source to t1.cpp. Let's take a look at t1.cpp:

```
[student@localhost test] less t1.cpp
# 1 "t.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "t.cpp"
# 1 "s.h" 1
void f();
# 2 "t.cpp" 2
```

```
int main()
{
    return 0;
}
```

Ignoring the lines beginning with the preprocessor directive symbol `#`, the source file `t1.cpp` looks like this:

```
void f();

int main()
{
    return 0;
}
```

(If you don't specify the output option, the preprocessed source is sent to the standard output, your console window.)

Get rid of `t.cpp`, `t1.cpp`, and `s.h` before going to the next section

Exercise 6.1. Produce a preprocessed source file from `helloworld.cpp`. Take a look at it! (You'll see why I did not use it as an example.) ☐

7 The Debugger Option

The GNU debugger, `gdb`, allows you to debug programs by for instance executing one statement at a time so that you can view the state of your program. Refer to the `gdb` tutorial for details.

In order for `gdb` to work, you need to compile your code with extra information. And to do that you need to include a special option. This is how you do it:

For a C++ program you do this

```
[student@localhost test] g++ -g helloworld.cpp -o helloworld
```

If you include the warn-all option, you get

```
[student@localhost test] g++ -Wall -g helloworld.cpp -o helloworld
```

For a C program you do this:

```
gcc -Wall -g helloworld.c -o helloworld
```

After the executable `helloworld` is created, you can run `gdb` on it:

```
gdb helloworld
```

Once you are confident that your program works correctly you should recompile without the `-g` option. Again refer to the `gdb` tutorial for details.

Besides gdb, there are other debugger tools that also requires or recommend the `-g` flag.

8 Object Files and Linking

Instead of generating the executable, let's generate the object files and then link them together. Before that let's get rid of the executables from previous sections:

```
[student@localhost test] rm a.out helloworld *.exe
```

Now do this

```
[student@localhost test] g++ -c helloworld.cpp
```

The `-c` option tells `g++` to compile object files and do not link. Do a list directory and you see that you have a new file:

```
[student@localhost test] ls -la
total 24
drwxrwxrwx. 1 root root  4096 Sep 17 17:08 .
drwxrwxrwx. 1 root root 16384 Sep 17 17:08 ..
-rwxrwxrwx. 1 root root   105 Sep 17 17:08 helloworld.cpp
-rwxrwxrwx. 1 root root  2712 Sep 17 17:08 helloworld.o
```

The `helloworld.o` is the object file produced from `helloworld.cpp`. If you want to change the name of the object file you can so this:

```
[student@localhost test] g++ -c helloworld.cpp -o someobjectfile.o
[student@localhost test] ls -la
total 27
drwxrwxrwx. 1 root root  4096 Sep 17 17:08 .
drwxrwxrwx. 1 root root 16384 Sep 17 17:08 ..
-rwxrwxrwx. 1 root root   105 Sep 17 17:08 helloworld.cpp
-rwxrwxrwx. 1 root root  2712 Sep 17 17:08 helloworld.o
-rwxrwxrwx. 1 root root  2712 Sep 17 17:08 someobjectfile.o
```

Usually you don't want to change the default name of the object file. Let's get rid of the second object file:

```
[student@localhost test] rm s*.o
```

And now to produce an executable from the object file (not the `cpp` file) do this:

```
[student@localhost test] g++ helloworld.o -o helloworld.exe
```

Do a list directory to verify that you now have an executable:

```
[student@localhost test] ls -la
total 46
drwxrwxrwx. 1 root root  4096 Sep 17 17:08 .
drwxrwxrwx. 1 root root 16384 Sep 17 17:08 ..
-rwxrwxrwx. 1 root root   105 Sep 17 17:08 helloworld.cpp
-rwxrwxrwx. 1 root root 22640 Sep 17 17:08 helloworld.exe
-rwxrwxrwx. 1 root root  2712 Sep 17 17:08 helloworld.o
```

Run `helloworld.exe` on your own to verify that you do have a valid executable.

9 Compiling multiple files

So far I've shown you the following: Given a single cpp file,

1. Build the executable
2. Build the object file and then the executable

Now let me talk about the case where you have multiple files, including header files as well. The process is very similar – so no big surprises.

Create the following files. Here's the `helloworld1.cpp`:

```
#include "print.h"

int main()
{
    print();
    return 0;
}
```

Here's `print.h`:

Here's the `print.cpp`:

```
#include <iostream>
#include "print.h"

void print()
{
    std::cout << "hello world" << std::endl;
}
```

I'm going to build the executable `helloworld1.exe` from `helloworld1.cpp`. I can do this by building the executable immediately:

```
[student@localhost test] g++ helloworld1.cpp print.cpp -o helloworld1.exe
helloworld1.cpp:1:10: fatal error: print.h: No such file or directory
    1 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
print.cpp:2:10: fatal error: print.h: No such file or directory
    2 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
```

It does work and for a small project this is probably best. In general if your project involves `a.cpp`, `b.cpp`, `c.cpp`, `d.cpp`, you can build the executable `e.exe` like this:

Of course if these are the only cpp files in the current directory you can also do this:

```
g++ *.cpp -o e.exe
```

where `*.cpp` means "all the files here that ends with `.cpp`".

But now I'm going to build the executable in two steps:

1. Build the object files
2. Link object files to get an executable.

You do have to type more commands. In the next section I'll explain why it's important to do it this way.

First, build the object files:

```
[student@localhost test] g++ -c helloworld1.cpp print.cpp
helloworld1.cpp:1:10: fatal error: print.h: No such file or directory
    1 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
print.cpp:2:10: fatal error: print.h: No such file or directory
    2 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
```

Do a list directory to check that we do have two new object files:

```
[student@localhost test] ls -la *.o
-rwxrwxrwx. 1 root root 2712 Sep 17 17:08 helloworld.o
```

Of course the above is similar to executing `g++` twice with the `-c` option:

```
g++ -c helloworld1.cpp
g++ -c print.cpp
```

Second, let's link everything to get an executable:

```
[student@localhost test] g++ helloworld1.o print.o -o helloworld1.exe
g++: error: helloworld1.o: No such file or directory
g++: error: print.o: No such file or directory
g++: fatal error: no input files
```

```
compilation terminated.
```

and vóila:

```
[student@localhost test] ls -la h*1.exe
ls: cannot access 'h*1.exe': No such file or directory
```

Run your executable `helloworld1.exe` to verify that it works.

Instead of the above command

```
g++ helloworld1.o print.o -o helloworld1.exe
```

you can do this:

```
g++ -o helloworld1.exe helloworld1.o print.o
```

They mean the same thing.

So let me summarize. If you have a project involving multiple C++ files, say `a.cpp`, `b.cpp`, `c.cpp`, `d.cpp`, you can build the object files like this:

```
g++ -c a.cpp b.cpp c.cpp d.cpp
```

or one at a time like this:

```
g++ -c a.cpp
g++ -c b.cpp
g++ -c c.cpp
g++ -c d.cpp
```

This will produce object files `a.o`, `b.o`, `c.o`, `d.o`. You then link them together

```
g++ a.o b.o c.o d.o -o e.exe
```

or this:

```
g++ -o e.exe a.o b.o c.o d.o
```

to get the executable `e.exe`. That's it! (for now ...) The process is the same if you have C program files except that you use `gcc` instead of `g++`.

Putting everything together to include the warn all and debug option (assuming you're still working/debugging the project), this is how you should use `g++`. Assume again that your project involves `a.cpp`, `b.cpp`, `c.cpp`, `d.cpp`. First, you generate the object files:

```
g++ -Wall -g -c a.cpp b.cpp c.cpp d.cpp
```

and second, then you link them:

```
g++ a.o b.o c.o d.o -o e.exe
```

to get the executable `e.exe`. If the files are C source files you use `gcc` instead of `g++`.

10 Why Build and Save Object Files?

If you execute

```
g++ helloworld1.cpp print.cpp -o helloworld1.exe
```

the compiler will build the intermediate object files (although they won't be saved so you won't see them in your project directory) and then link them together to get the executable `helloworld1.exe`. The object files are not saved.

So why is it important to manually build the object files on your own so as to have the object files saved in your project directory?

Because in real world programs, a C++ source file might be huge and it can take a long time to generate the object file. Further, a complete C++ project might involve a huge number of C++ source files. Suppose you have a C++ project in a directory and the project involves 1000 C++ cpp files, including the main C++ source file, say `game.cpp` which containing the `main()` function; say all the relevant header files are also in the same directory. `game.cpp` uses the other 999 C++ files directly or indirectly.

Now suppose you modified two files. Obviously you want to rebuild your executable and test your program. So you execute

```
g++ *.cpp -o game.exe
```

`g++` would recompile all the 1000 C++ files to obtain 1000 new object files and then link them together (your object files are not saved.)

However if you built object files separately so that the object files were kept in your project folder, then you only need to rebuild the object files for the modified files or those that uses the modified files directly. If the modified C++ files are

- `physics.h` and `physics.cpp`
- `sound.h` and `sound.cpp`

then you can do the following:

```
g++ -c physics.cpp
g++ -c sound.cpp
g++ -c game.cpp
g++ *.o -o game
```

assuming `game.cpp` is the only cpp file that uses `physics.h` or `sound.h`. This dramatically saves you a lot of time when compared to building 1000 object files. The linking process would still take the same amount of time.

In general, you have to rebuild object file `x.o` if

1. you modified `x.cpp` or
2. `x.cpp` contains `#include "y.h"` and you modified `y.h`

Although you need to know all the benefits of saving object files, remember that the type of projects you will work on right now will be small (unless you are working on some humongous personal project). So you won't really see the benefit in a dramatic way. But even if you're working on a small project, but the project uses the source files of some really huge project, then it will have an impact. In the real world, recompiling a whole C++ project can take hours.

11 Include Directory

It's possible to tell g++ to look for header files other than in the directory containing your main program. This is helpful for instance if you're reusing code from another project.

Suppose your current directory has `helloworld1.cpp` (as before):

```
#include "print.h"

int main()
{
    print();
    return 0;
}
```

Let say `print.h` is somewhere else: create a directory inside your current directory, say `include`, and move `print.h` into that directory:

```
[student@localhost test] mkdir include
[student@localhost test] mv print.h include
mv: cannot stat 'print.h': No such file or directory
```

Now if you try to compile your program, g++ will complain:

```
[student@localhost test] g++ helloworld1.cpp -c
helloworld1.cpp:1:10: fatal error: print.h: No such file or directory
    1 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
```

Why? Because for your header files, g++ will only look at your current directory. (g++ will look for standard header files such as `iostream.h` at special system level directories – you don't have to worry about such header files.)

Now try this:

```
[student@localhost test] g++ helloworld1.cpp -c -I include
helloworld1.cpp:1:10: fatal error: print.h: No such file or directory
    1 | #include "print.h"
      |          ^~~~~~~~~
compilation terminated.
```

It works! Why? Because the `-I` option tells g++ to look for header files in the `include` directory. Of course if your header files are in directory `/a/b/c/d`, then you execute

```
g++ helloworld1.cpp -c -I /a/b/c/d
```

Exercise 11.1. What if you want `g++` to look in *two* different directories for header files?
□

By the way, what if you need to link with an *object* file that sits in a different directory? Say I move `print.o` to a new directory `obj`:

```
[student@localhost test] mkdir obj
[student@localhost test] mv print.o obj
mv: cannot stat 'print.o': No such file or directory
```

Then I just specify the path to the object file:

```
[student@localhost test] g++ helloworld1.o obj/print.o -o helloworld2.exe
g++: error: helloworld1.o: No such file or directory
g++: error: obj/print.o: No such file or directory
g++: fatal error: no input files
compilation terminated.
```

12 Libraries

I started with directly compiling one or more C++ source files directly to an executable, following by compilinng object files from the course files and then linking them into an executable. In the previous section, I talked about how to tell the compiling to look for header files elsewhere – this is when the project gets huge and either you want to put your files into multiple directories or you want to reuse the code from another project at another directory.

Besides the header files being stored somewhere, of course the object files can be somewhere else too. As mentioned in the section on header files, you just need to specify the path to the object files if the object files are not in the current directory.

In the linking process, you combine object files. Beisdes object files there are also "library files" which are like object files. For instance you have been using the functions defined in the header `iostream.h`. Obviously your executable must somehow be build by linking with the library code associated with `iostream.h`. The library is called `libstdc++.so`. In the case of `libstdc++`, you do not need to tell `g++` to use it – it's automatic.

For non-standard llibraries, you have to tell g++ to link to the libraries. For instance

```
g++ main.cpp -lSDL -lpthread
```

In this case, I'm telling `g++` to link with the `libSDL.so` and `libpthread.so` library.

Note that if you want to use `libSDL`, in the `-l` option you write `SDL` and not `libSDL`.

Note that the libraries mentioned above are obviously not in your directory. Where are they? They are usually found in `/usr/lib/`. You can also specify a library path using the `-L` option.

13 Optimization

Once you're really certain that your program works correctly you can optimize it. First you do not want to include the debug option. This will prevent extra debugging information from being included in your executable. Second you can include the optimization option. This is how you do it:

Let's say you have a `main.cpp`. You also have a class made up of `foo.h` and `foo.cpp`.

```
g++ -O2 -c foo.cpp
g++ -O2 -c main.cpp
g++ -O2 -o main foo.o main.o
```

If the files are C source files (and they end in `.c` instead of `.cpp`) then you can also do this:

```
gcc -O2 -c foo.c
gcc -O2 -c main.c
gcc -O2 -o main foo.o main.o
```

There are actually many optimization levels (`-O`, `-O2`, and `-O3`). Refer to man pages for details.

14 Summary

Your basic g++ command should probably be

```
g++ *.cpp -g -Wall
```

This should be sufficient for most assignments.

15 What now?

`g++` is an extremely complicated piece of software. Remember that you can find lots of information just by using the man pages:

```
man g++
```

The number of options can be overwhelming. Another route to learning more about `g++` is of course to use the web and search for `g++` tutorials.

You now know how to go from C++ source file(s) to executable (and how to run executables) and from source files to object file to executable. The next thing to learn is how to manage your collection of object files. Recall that in the section on object files, you only need to rebuild an object file if the corresponding C++ source file is modified or if the C++ source file uses a modified header file. It turns out that you don't have to worry about remembering which files were changed or which object files need to be recompiled. Linux has tools to manage that for you. There are many such software in Linux to handle this aspect of project. The simplest and most popular is called `make`. Once you've specified the "relationship" between files in your project, when you run `make`, it will decide which object file has to be rebuilt.