Building C++ Files

Here are some short notes on building C++ applications. C++ files need to be compiled into binary form before execution. On Linux we normally use the Gnu compiler. This compiler is also used to compile C files and therefore Gnu provides two different commands: **gcc** for compiling C programs and **g++** for compiling C++ programs. Having two different commands allows the Gnu compiler to automatically link to the C runtime libraries in the case of gcc and the C++ runtime in the case of g++. We will be using g++ throughtout this course.

The g++ command actually performs preprocessing, compilation, assembly and linking on a given source file.

In recent years, Microsoft, Apple, Google and others have developed a drop-in replacement for the Gnu complier, called **clang**. Again, this comes in two forms: **clang** and **clang++**. You can use this compiler during the course and all examples should work just as well.

Suppose we have a C++ source file called **example.cpp**; we can compile and link this file with:

```
g++ -g example.cpp -o example.exe
```

This will create an executable file called **example.exe** as specified by the **-o** option. If we intend to debug this program we need to use the **-g** option. This instructs the compiler to include a symbol table with the executable code. This will useful for debugging, but undesirable for production code. You can inspect this symbol table with the **nm** command:

```
nm example.exe
```

Of course, real systems will comprise of many source files and each needs to be processed by g++. With a large system comprising of many files this can take a long time. In practice, during development, this build process is undertaken many times. Usually only a few source files get changed between builds and it would be overkill to keep rebuilding source files that had aleady been compiled. To overcome this problem, the **make** utility was developed. Before we look into make, we need to take a short digression to see how large systems are built.

The compilation process for large systems is somewhat more complicated than single source file systems. In this case, C++ source files are compiled into relocatable object files (binary, but not fully converted to executable format). When all source files have been converted into relocatable object form they are then linked together to form an executable file. The g++ command is used for both the compiling and seperate linking process using different options.

Suppose we have a C++ project consisting of 3 files:

```
example1.cpp
example2.cpp
example3.cpp
```

We can create 3 relocatable object files using the **-c** option with g++:

```
gcc -g -c example1.cpp
gcc -g -c example2.cpp
gcc -g -c example3.cpp
```

The relocatable object files conventionally have a .o extention, so this will produce:

```
example1.o
example2.o
example3.o
```

The relocatable files can be combined into an executable file by linking with g++:

```
g++ example1. example2.o example3.o -o example.exe
```

When you supply just .o files to g++ it knows you just want to perform linking (the compilation has already been completed).

Using Make

When a source file is compiled into a .o file the timestamps on the two files will show that the .o file was created after the .cpp file. However, if the source file has been edited, but not yet compiled, the timestamps will be reversed such that the .cpp file is more recent than the .o file. The make utility makes use of these timestamps to work out which files need to be recompiled and which files have already been compiled and can be exempted from the build process. This can save an enormous amount of time on large builds.

The Gnu make utility (**gmake**) uses a 'makefile' that contain all the dependencies and rules for building the system. In the 3 file example given above, we know the .o files are dependent on the .cpp source files. The rule for building is given above using g++ -c. This is encoded in the makefile with:

```
example1.o : example1.cpp              # dependency
        gcc -g -c example1.cpp         # rule (must be have a tab as first character)
example2.o : example2.cpp
        gcc -g -c example2.cpp
example3.o : example3.cpp
        gcc -g -c example3.cpp
```

But we have a further dependency and rule for the final linking:

```
example.exe : example1.o example2.o example3.o
        g++ example1. example2.o example3.o -o example.exe
```

The rules and dependencies are gathered together in a file called **Makefile**:

```
example.exe : example1.o example2.o example3.o
        g++ example1. example2.o example3.o -o example.exe
example1.o : example1.cpp                    # dependency
        gcc -g -c example1.cpp               # rule (must be have a tab as first character)
example2.o : example2.cpp
        gcc -g -c example2.cpp
example3.o : example3.cpp
        gcc -g -c example3.cpp
```

Now when we issue the command:

```
make
```

the utility checks if the first target (example.exe) has been built using the appropriate timstamps. If any of the .o files are out of date (older than the .exe file) then make invokes the rule to build that .o file.

Of course this is a trivial makefile. In practice makefiles on large project usually end up as quite complicated.