ECE244 Programming Fundamentals

Lab Assignment #2: The `Make` Utility

1. Objectives

The objectives of this assignment are to introduce you to the use of the `Make` utility.

2. Problem Statement

In this lab, you will practice the use of the `Make` utility to compile programs.

3. Background

In practice, a program is expressed in many source files (numbering in thousands for large programs). All these source files must be compiled and the resulting object files linked together to build the executable. As the program is developed, and its source files are changed, the modified sources must be re-compiled and the resulting new object files re-linked to produce a new executable that reflects the changes made to the sources.

However, it is often the case that changes are made only to a small number of these source files at any given time. While it is possible to re-compile all the source files to re-generate the executable, only the sources that changed really need to be re-compiled. Indeed, re-compiling all the source files when there are thousands of them is very inefficient and very time consuming. Keeping track of which files changed and need to be re-compiled can be a daunting task. The `Make` utility helps a programmer by automatically figuring out which source files changed since the executable was last built, and automatically re-compiling them. The programmer only has to express how the executable depends on the various source files (called make dependences) in a file called the `Makefile`. `Make` takes care of the rest.

4. Preparation

There is a tutorial on how to use the `Make` utility in the "Labs" section of the course's web site. Read through the tutorial ***before*** you go to the lab!

5. Procedure

Create a sub-directory called `lab2` in your `ece244` directory, using the `mkdir` command. Make it your working directory. Make sure it is accessible only by you using the `chmod` command.

Use a browser to download the files `Main.cpp`, `squared.h`, `squared.cpp`, `squareRoot.h`, and `squareRoot.cpp`. Examine the files. They collectively implement a very simple (almost trivial) program that prompts the user for a number and then prints the square and the square root of the number. The functionality of the program is spread in three files. `Main.cpp` contains the `main` function, which performs I/O and calls the two functions `squared()` and `squareRoot()`. The files `squared.h`, and `squared.cpp` are respectively the definition (or declaration) and implementation files for `squared()`. Similarly, `squareRoot.h`, and `squareRoot.cpp` are respectively the definition and implementation files for `squareRoot()`. Observe how the definition files of the two modules are included in the various `.cpp` files.

To generate an executable for this program, each source files is compiled separately to generate an object file, and then the object files are linked together.

First, compile `Main.cpp` to produce `Main.o`:

```
g++ -c Main.cpp
```
(command 1)

Second, compile `squared.cpp` to produce `squared.o`:

```
g++ -c squared.cpp
```
(command 2)

Third, compile `squareRoot.cpp` to produce `squareRoot.o`:

```
g++ -c squareRoot.cpp
```
(command 3)

Finally, link the three object files `Main.o`, `squared.o`, and `squareRoot.o` to produce the executable `Main`:

```
g++ Main.o squared.o squareRoot.o –o Main
```
(command 4)

You may want to list your directory (`ls` command) after executing each of these commands to see the object files generated. Execute `Main` to see the program working.

The above commands expose the dependences that must be expressed in a `Makefile`. To generate the executable, the three object files `Main.o`, `squared.o`, and `squareRoot.o` must be linked together (*command 4*). If any of the object files changes, this command must be re-issued to re-generate the executable. Thus, we say that `Main` *depends* on `Main.o`, `squared.o`, and `squareRoot.o`. Similarly, to generate `Main.o` the file `Main.cpp` must be compiled (*command 1*). If `Main.cpp` changes, then *command 1* must be re-executed to re-generate `Main.o` (which in turn causes *command 4* to be re-issued to re-generate the executable). The same holds for `squared.o` and `squareRoot.o`.

Now that you have determined the commands necessary to compile the program, you are ready to write the `Makefile`. Create a new file called `Makefile` and enter the following lines in it. The → symbol indicates that the spaces you see are generated with a single tab character, not space characters.

```
Main:          →    Main.o squared.o squareRoot.o
               →    g++ Main.o squared.o squareRoot.o -o Main

Main.o:        →    Main.cpp squared.h squareRoot.h
               →    g++ -c Main.cpp

squared.o:     →    squared.cpp squared.h
               →    g++ -c squared.cpp

squareRoot.o:  →    squareRoot.cpp squareRoot.h
               →    g++ -c squareRoot.cpp
```

The line "`squareRoot.o: → squareRoot.cpp squareRoot.h`" expresses the dependence of `squareRoot.o` on `squareRoot.cpp` and `squareRoot.h`. If any of these two files change, then `squareRoot.o` must be re-generated. The command to do so is given in the line below as: "`g++ -c squareRoot.cpp`", which is *command 3* above. Similarly, the line "`Main: → Main.o squared.o squareRoot.o`" expresses the dependence of the executable `Main` on the object files `Main.o`, `squared.o`, and `squareRoot.o`. If any of these objects change, then the executable `Main` must be re-generated. The command to do so is just underneath: "`g++ Main.o squared.o squareRoot.o -o Main`", which is simply *command 4*.

You may now experiment with the `Makefile`. First, cleanup the directory by deleting all the object files and the executable you generated earlier (***<u>be careful</u>***):

```
rm –i *.o Main
```

Now, type `make` at the command prompt, and observe the commands that get executed. Verify that all the files are compiled and the executable is made.

Now you will experiment with making changes to the various source files and re-typing `make` to find out what gets re-compiled. To make things simple, you will only make changes to the comments in the various source files and save the changes. Alternatively, you can update the timestamp of a file using the `touch` command (`man touch`). The remainder of the assignment refers to either of these options as ***<u>updating</u>*** the file.

Update the file `Main.cpp` (i.e., make a simple change to the comments in the file and save the file, or issue the command `touch Main.cpp`. Re-type `make`. What commands get executed? Do all the source files get re-compiled? If you redo these steps, does the order in which `make` executes its commands remain the same?

Now update the file `squared.cpp`. Re-type `make`. What commands get executed? Observe that not all the source files get re-compiled. Only those that are affected by the change (i.e., `squared.cpp` and `Main`) are re-compiled.

Now update the file `squareRoot.cpp`. Re-type `make`. What commands get executed? Observe that not all the source files get re-compiled. Only those that are affected by the change (i.e., `squareRoot.cpp` and `Main`) are re-compiled. Why is it necessary to re-link the objects?

Now update the file `squared.h`. Re-type `make`. What commands get executed? Do all the source files get re-compiled? Why is `Main.cpp` re-compiled?

In all the above examples, make sure that you understand why `make` executes certain commands while skipping other commands. Also, in each example, can you explain why `make` executes the commands in a specific order.

6. Deliverables

Submit all components of this assignment, including the `Makefile` you wrote. Namely, `squared.h`, `squareRoot.h`, `squared.cpp`, `squareRoot.cpp`, `Main.cpp` and `Makefile` using the `submitece244s` command as follows:

```
submitece244f 2 squared.h squareRoot.h squared.cpp
                                squareRoot.cpp Main.cpp Makefile
```