COMP161 - Lab 2 & Homework 2

Spring 2016

These assignments are designed to get you working with the GNU GCC, C++ compiler **g++**, the build scripting program *make*, and some features of Emacs that make programming easier. For homework, you'll be directed to "break" the code in order to see how the compiler reacts to some common mistakes.

Overview

You should give Lecture Notes 5 at least a quick scan read prior to starting this lab as they pretty much layout everything you need to do. The code you'll be working with for these assignments can be found at */home/comp161/sp16/* on the server. The four files are zipped up in *lab2.zip*.

Lab 2

In lab you're pretty much working through the lecture notes to get practice building and running programs and tests. A few times you'll be asked to redirect the output of some commands to a text file. This is meant to give you practice with some more advanced CLI features and to give me evidence of your success and progress.

Now, carry out the following tasks[1]:

1. Copy the file *lab2.zip* from */home/comp161/sp16* to your home directory. Use the command *unzip* to extract the files.[2].

2. Use *g++* to compile each of the cpp files into objects files.

3. Use *g++* to build the main program executable. Name it *fact_main*.

4. Use *g++* to build the test executable. Name it *fact_tests*.

5. Run the main program a few times with a few different arguments in order to get comfortable running executables that you yourself built.

6. Run the test executable with the option to list all the tests. Use the CLI to redirect this output to a file called *lab2-testOutput.txt*.

7. Run each of the test *cases* individually. Use the CLI to append the output to the test output file you started in the previous task.

8. Run all of the individual tests from one of the cases that has multiple tests. Again, use the CLI to append the output to your test output file.

[1] Details on each can be found in Lecture Notes 5

[2] use *man unzip* to get the manual for unzip and see if you can figure out how to use it from that

9. Write a Makefile for this program. Write it such that running the command *make* without arguments will build both the main and test executable. Do not copy and paste text from the notes. Type it out by hand to get practice working with Emacs. You might consider doing one rule at a time and testing it after each new rule is added.

10. Clean out all the objects and executables using make. Then use make to compile only the test executable. Redirect the output of a sucessful, clean make of the test executable to a file called *lab2-builds.txt*[3].

11. Clean your build files up again and do a clean build of just the main executable. Use the CLI to append the output of this make command to your build file started in the previous step.

12. Do a clean build of the make all rule and append that output to the build file.

You should now have two text files containing redirected CLI output: one from running Google Unit Tests and the other from make. You should also have a Makefile that you created from scratch. Submit these three files and *only these three files* using *handin*. If you submit the C++ code, executables, objects, or anything not one of these files, then you'll lose points for the lab. The assignment is *lab2*. To submit more then one file with handin you simply need to group those files in a single directory and submit that directory. The handin output should report which exact files were submitted.

Homework 2

Due by *class* Monday 1/25

In this homework you'll be introducing errors to the code and seeing how the compiler reacts. While doing this, you should practice building code from within Emacs.[4]. For each error you introduce you need to carefully examine the compiler's error report and write a brief statement, a few sentences, of how it seems to relate to the error as you see it. Write this report up in Emacs[5] and submit it, and only it, via *handin* as assignment *hwk2*. Some error messages are better than others. When they seem to be totally unrelated, take some time to really examine and analyze them for some clue about the error.
Here are the errors you should introduce:

1. *No main procedure*

   Compile an executable without a *main* procedure. There are two easy ways to do this by modifying your Makefile instead of the

[3] By clean I mean starting with only source code files and no pre-compiled objects or executables

[4] I highly recommend you build code within Emacs at least once while in lab so you can get help if you need it

[5] Don't forget your name!

code itself. If you remove the *-lgtest_main* from the rule for the test executable, then you'll be missing the Google provided main. If you remove the lab2_main.o from the compile command for the main executable, then you're missing the program main. I suggest you try them both.

2. *Missing Semi-colon*

   In *factorial.cpp*, remove the semi-colon at the end of line 42. Any line ending in a semi-colon is a C++ statement. This particular statement instructs the computer to call another function. You might try removing other semi-colons to see if the different statements induce different errors when they're missing their terminating semi-colon.

3. *Missing namespace specifier*

   The statement on line 18 of *lab2_main.cpp* directs the computer to look in the *std* namespace for names it doesn't known. This allows us to more easily call several functions from standard C++ libraries. Remove the code on line 18.

4. *Missing #include statement*

   In *lab2_main.cpp*, remove lines 10-13, each individually. Each line includes a different library. By doing each line individually you can see similarities and differences between forgetting to include different libraries.

5. *Missing define guards*.

   In *factorial.h* you should see the define guard code[6] on lines 10, 11, and 103. Remove all of those lines at once to get rid of the define guard. Now do complete, clean build of the project, i.e. testing and main executable.

   [6] See lecture notes 4

6. *Choose your own adventure*

   Come up with at least two more errors to introduce. You might try searching around the web for common C++ syntax mistakes and see if you can introduce one of those. Coming from Racket, an obvious thing you could do are missing parenthesis and curly braces. Just like in Racket, C++ makes use of matching opening and closing parenthesis and brackets quite often. When you type a closing bracket/parenthesis in Emacs, it will briefly highlight the opener it is associated with. If you're not sure what brackets go with what, then delete and retype a closer to see what opener it goes with. Try removing an open and a closer. Try different situations; not all brackets are created equal. Some brackets surround definitions where others are used to block statements together to make another statement within a definition.

C++ Comments

When introducing errors, I recommend that you comment out code rather than delete it. So when I say remove, I really mean comment out. For more involved errors, it'll be easier to remember to remove comment markers that you introduced than to re-type code that you didn't write and have never written!

There are two ways to comment things out in C++: `//` and `/∗ ∗/`. The double forward slash will comment out everything after it until the end of the line; it's a single line comment. The matching `/∗` and `∗/` create a block comment. Everything between them is commented out. You should see examples of both comment styles in the code.

Emacs and plain text

If you're writing plain text with emacs then there are two things you need to be aware of: controlling the width of the text and spell-check. If your text is too wide, spans too many columns, then it will print funny. This is easily avoidable using fill commands[7]. If you're like me, you're too reliant on spell-check. Emacs integrates with a linux-based spell checker and provides some convenient commands to check your spelling[8]. *Keep your document to a width of 60 characters and use spell-check.*

[7] http://www.gnu.org/software/emacs/manual/html_node/emacs/Fill-Commands.html

[8] http://www.gnu.org/software/emacs/manual/html_node/emacs/Spelling.html