COMP161 - Homework 2 & Lab 2

Spring 2014

These assignments are designed to get you working with the GNU GCC, C++ compiler **g++**. You'll be working with pre-written C++ code so that you're free to focus on the compiler and it's different usages.

Homework 2

Due before lab, Wednesday January 29.

For this homework assignment you need to complete the following tasks. Some more detailed instructions are given below the task list.

1. Read the first two sections of Lecture Notes 05[1]
2. Copy the file *comp161-lab3.zip* from */home/comp161/sp14/*. Unzip the file with the command *unzip comp161-lab3.zip*. The files contained in this zip file are described in lecture notes 05.
3. Compile the main program as directed in the lecture notes.
4. Use the program to compute the factorial of 1,5,10,15, and 20 and gather performance data about the execution of the program[2]. The results of all 20 executions of the program should be appended to a text file using the appropriate bash redirect.[3]
5. In the file containing your performance data, answer the following questions:
   (a) Compare and contrast the running times of the different versions. Is there a clear winner in terms of speed? What do you make of these results?
   (b) Compare the computed results to the actual values[4]. What do you see? Now go look up *integer overflow*. What do you make of this?
6. Submit your text file as assignment *hwk2* using *handin*.
7. Finish going over lecture notes 05 in preparation for lab 3.

[1] Up to and including *Basic Compilation*

[2] See below for details on using *time*

[3] Be sure the text file has an appropriately descriptive filename, contains your name at the top, and has the *txt* file extension.

[4] See the factorial entry on wikipedia or google it

Running the Factorial Program

The program takes two inputs. The program computes the factorial of the first using a version of the factorial computation dictated by the second. The program attempts to error check your input, but the first number should be a postive integer or zero and the second should be 1,2,3, or 4. So, if the executable you compile is named factorial, then:

```
./factorial 5 3
```

will compute 5! using version three of the factorial code.

Timing the execution of a command

There's a command called *time* that will time the execution of a command and report that time back to you.[5]. To time our earlier factorial execution we simply do the following:

[5] check out the man page for more details

```
time ./factorial 5 3
```

The time data is, by default, written to the standard output along with any other information written by the program being timed. You'll see three time values: real, user, and system. Focus on real time, but see `http://stackoverflow.com/a/556411/1042494` for an excellent break down of the details of each time value.

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[6]. 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[7]. *Keep your document to a width of 60 characters and use spell-check.*

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

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

Lab 3

For this lab you'll continue playing with the compiler as well as learning how to run gTest unit tests. As you do you'll be asked to record some data to a text file like you did for the homework[8]. In the course of the lab you'll be asked a few questions, respond to those questions[9] in the same file as your data. At the end of lab, submit your text file (and nothing else!) as *lab3* using the *handin* command.

[8] Once again, give your file an appropriately descriptive name, put your name at the top, and use the *txt* file extension

[9] use full sentences!

1. Open *factorial.cpp* with emacs. What do you make of the code? What's your initial impression of C++?
2. Compile each cpp file doing preprocessing only and write the results to a file[10]. Use the command *wc*[11] to get some size statistics about the size of the original file as well as the file that results from preprocessing. Use CLI redirects to append all the *wc* data to your text file.
3. Which files get bigger as a result of preprocessing? Which grows the most?
4. Open the preprocessed version of *factorial.cpp*. What do you make of this code? How does it compare to the original?[12]

[10] can be done with g++ or using redirects

[11] see below

[12] Use emacs windows to open both files side by side

5. Now compile each cpp file using the preprocessor and compiler only to get the assembly code. Once again, use *wc* to record some basic data about the size of the assembly code files and write your results to your text file.
6. What's larger the preprocessed C++ or the assembly?
7. Open the assembly version of *factorial.cpp* with emacs. What do you make of assembly code?
8. Now compile each cpp file down to an object file[13]. Once again, use *wc* to get size data and record your data to your file.
9. How does the size of the object file compare to the other versions of the file?
10. Open the object file for *factorial.cpp* with emacs. What do you make of that stuff?
11. Compile the gTest main program as described in the lecture notes.
12. Run the program with the option to print all the test cases and names and write the results to your text file.
13. Now, run each test *case* individually and write the results to your file.
14. Compile *factorial.cpp* down to assembly code using each of the three optimization levels. Use *wc* to gather size data about each version.
15. How do the sizes of the four versions[14] compare? What do you make of this?
16. Compile a version of the main program for each optimization level. Be sure to name each version something different.
17. Compute the factorial of 10 with each of the four versions of the computation and using all four optimization levels for each version. Use *time* to record timing data for all 16 computations and add this data to your text file.
18. What do you make of the results your timing experiments?

[13] i.e. do everything but link

[14] original and three optimization levels

If you've completed all the tasks listed above, then you're done. Submit your text file. If you're looking for something else to play with, start deleting stuff from *factorial.cpp* and compiling the file to see what kind of error messages arise. Be systematic with what you delete and experiment to see if you can start getting a sense what typos produce what errors.

The word count (wc) command

The command *wc* counts newlines, words, and bytes in a file[15]. Focus your attention on the third number reported by wc, the bytes.

[15] As always, see the manual page for more details