Lab 1 - Setting Up Your Environment

Due Date: 5:00 p.m., February 10, 2015
All function interfaces are suggested naming and parameter guidelines. If you feel there is a better way, you are free to alter names, functions interfaces, etc, as long as you follow the lab and style guidelines. Output should match exactly unless otherwise stated.

For the first part of lab we are going to log into our account and write a 'Hello World' program in C++. This will ensure everyone has access to their accounts and all of the tools needed for the course.

(Parts A and B must be completed in lab)

Part A: Learning the Environment and Writing the code

Log into a machine using your CS Department userid and password. (If you're reading this online in the lab, you've probably succeeded in doing this already.) Locate and launch a "terminal". We will call this, inter-changeably, a terminal, shell, or command line. The shell will accept and execute linux commands, of which there are hundreds. We'll learn a few today, and I'll try to introduce you to more and more throughout the semester.

The Linux help system is accessible through the system man pages. This is standard on all Unix/Linux distributions. "man" is short for "manual" (but the command is "man"). To access the man pages for a particular command, at the command prompt type the command "man" (without quotes) followed by the command you are interested in, followed by the <ENTER> key.

- For example, to read about the "pwd" command, type ‘man pwd’
    - This will open up the man program and display the help information for the pwd command. In the man page environment, to go forward and look at additional information that does not fit on the first screen, press the <ENTER> key. To exit the man page program, type the letter q. We will be using the following commands often, so review them in the man pages:
        - man
        - pwd
        - mkdir
        - rmdir
        - cd
        - ls
        - mv

- g++
- make

Using Linux Commands

Next, you will begin running and using some of the commands you learned about in the previous section above. In particular, complete the following steps:

- Create a directory named cs240. On the command line, you can do this with the following command:
    - mkdir cs240
- Change your "current working directory" to be the cs240 directory. To do this, execute the following command:
    - cd cs240
- Confirm that you are in fact in the cs240 directory by typing the command:
    - pwd
- Create a directory within cs240.... name it lab1. "cd" into this new directory.

Compiling a C++ Program

So far, you have not done any "programming" or even dealt with any C++ code. You have run a few simple linux commands, familiarized yourself with the linux environment, and learned how to create and navigate directories. In the next few sections, you will edit, compile, and run a C++ program within linux.

To create a file, you will need to use a unix file editor. Unix systems generally have several editors from which to choose. Two of the more popular and classic Unix editors are called “vi” (or “vim”) and “emacs”. Another popular editor is "nano". You may use any editor you prefer, but if you are learning one for this class, I don't recommend vi or emacs (yet). Nano is slightly simpler, but still not the best choice....

If you are not already very familiar with a linux editor, please use gedit, which has several nice features for C++ code development. You can use gedit to open a file with the appropriate name for your definitions, by typing, for example, the following into a linux command shell:

```
gedit McGillicuddy_Fritzy.cpp
```

If there is no file in the current folder named McGillicuddy_Fritz.cpp, then gedit will create a new empty one for you. After making changes to a file, you can save it by clicking "Save" at the top of the gedit window, or by typing <Ctrl-s>. Note that if you have made changes to an open file, and those changes have not been saved, the file name across the top of the window will be preceded by an asterisk (for example, “*McGillicuddy_Fritzy.cpp”). To exit from gedit, make sure your modifications are saved, and then type <Ctrl-q> in the gedit window, or simply close the gedit window by clicking the “x” in the upper right hand corner.

The only problem with gedit is that it will not work if you are trying to program remotely, from a machine other than the ones in Q22 or G7. To work remotely, you can use an editor (such as nano) that works within a shell, or you can edit the file on another machine and transfer it to the CS file server separately. See the remote access link for more information about working remotely.

You should be able to learn enough about text editing and working remotely on your own, or from a classmate, for this class. Please let us know if you need help.

- Create a file named lab1.cpp within the Lab1 directory.
- Create a hello world program exactly as we did in class using the following line to print out ‘Hello World’:
    - cout << “Hello World!” << endl;
- Don’t forget you will need to include the iostream library:
        - #include <iostream>
- Save and exit.

Part B: Compiling and Executing

- Like you did in the previous step, navigate to your lab folder in the shell.
- Type in 'ls' to list the directory's files. Ensure that 'lab1.cpp' is in the directory.
- To compile, we will be using the program called 'make'. Do not confuse this with the 'makefile' which we will be creating later. 'make' is the program that reads makefiles.
- As described in class, create a makefile to compile your lab1.cpp source code to an executable called ‘lab1’.
    - To compile the C++ source file, execute the following command:
        - make
    - This command produces an executable file named “lab1” by executing the g++ command found within the contents of the makefile. In particular, the “make” command looks for a file named “makefile” in the current working directory, and finds within that file the rule for “all” by default. This rule indicates that “all” should follow the rule for "lab1", contained below it. The lab1 rule, in turn, contains the following two lines:

        -
          ```
          lab1: lab1.o
          	g++ lab1.o -o lab1
          ```
    - You could also target specific rules
        - make lab1
    - The source should compile to an executable. You can run the executable with the command, './<executable>. In this case, './lab1'

Importantly, there is a [tab] character (not a single space) before the g++. These two lines indicate that the lab1 target depends on the lab1.o file. This causes the lab1.o rule (also to be defined within the makefile) to be invoked. The lab1.o rule depends on lab1.cpp. So if lab1.cpp has been updated recently (i.e. since the time that lab1.o was created), then the make utility will re-run the command that follows (in this case “g++ -c lab1.cpp”). This line (“g++ -c lab1.cpp”) is a linux command, and could be executed from a shell prompt. In our case, it is being executed by make, to compile the C++ code contained in lab1.cpp. The command's output is a file named lab1.o, which contains object code for the Hello program. This object code is then linked into an executable file with the command under the rule in the makefile for the “lab1” target. g++ also does this linking, this time via the following command:

- g++ lab1.o -o lab1

To repeat (in an attempt to be as clear as we can), g++ is the compiler and the linker... it's doing all the work of building your C++ program into object code and an executable that can be run over linux. The makefile is not strictly necessary for building C++ programs; it just makes things more convenient. The makefile contains rules for the linux make utility to invoke g++ so that it can do its thing. This way of building programs is intended to give you control over the build process, to set it up however you want, without requiring you to type in a bunch of detailed commands every time you compile your code. (That is, once everything is set up properly, you just type “make”, instead of all the different commands that are contained in the makefile.)

To verify that the two new files (lab1.o and lab1) were created, run the following command:

- ls
    - (As you know, "ls" lists the files in the current working directory.)

The final thing to make sure you have in the makefile is a rule for "make clean," which should run runs the following command:

- rm -f *.o lab1

This rm command removes all files that end in a ".o" extension (* is a wildcard character on a linux command line), and also removes the file "lab1"). Run "make clean" and then "ls", and observe that the files that make (via g++) had created earlier are now gone:

- make clean
- ls

To rebuild your program and get them back, run make again:

- make

Object files and the executable are not rebuilt if the files that they depend on have not been changed. Running and Updating a C++ Program

So far, you have only compiled and linked a C++ program, but you have not run it yet. To run the program, type the following at the shell prompt:

- ./lab1
    - In linux, "." is a shortcut name for the current working directory, and ".." is a shortcut name for the directory just "above" it.

Observe the output of the program, and then look in the lab1.cpp file to see the program that produced it. To look in the lab1.cpp file, you can edit it (with an editor described above), or you can use 'cat', 'more', or 'less' (all linux commands).

Altering and Recompiling a C++ Program

Now, let's change the C++ code and recompile and rerun it. Open the lab1.cpp file and add another line to the code to have it print "C++ Data Structures!". Immediately upon doing so, save the file and try running the program again:

- ./lab1

Notice that your new message is not printed. This is because you only updated the C++ file, not the executable file (lab1). To rebuild the executable, run make again. Now run lab1 and observe that it prints your new message.

Show your TA your code, and that you can compile your code with a makefile.
--END OF IN LAB REQUIRED WORK--

Part C: Separating C++ Code into Multiple Source Files

Next, we are going to create a class, and use separate compilation to compile everything.

- Please create two new new source files, called Hello.cpp and Hello.h.
    - Leave the main() function in lab1.cpp.
- You will need to create a class definition and implementation.
    - Your class class definition should go in Hello.h and should should contain a method
        - Your class definition should look like the following

            -
              ```
              class Hello{
                      public:
                              void printHello(void);
              };
              ```
        - You will also need include guards
            - (top of the file)
              ```
              #ifndef HELLO_H
              #define HELLO_H
              ```

            - (bottom of file)
              #endif
    - Your class implementation should go in Hello.cpp, and should define the following method
        - void Hello::printHello(){
                    cout << “Hello from the Hello Class!” << endl;
          }
        - This means you will need to include the iostream library in hello.c
            - #include <iostream>
    - Lastly, update your lab1.cpp to use your new class.
        - In your main, create a hello object from your Hello class
            - Hello hello;
        - Call your printHello method
            - hello.printHello();

Notice that the Hello class contains function definitions along with code that should be executed when the functions are called. Better practice (at least when classes grow larger) is to keep only the function interfaces (function name, return value, and parameter types) within the class definition, and to move the function bodies outside of class definitions. The syntax for doing so is as follows:

- Sample member function definition within a C++ class:
    - int myFunction(char *, string, int);
- Sample corresponding member function outside of the class definition:
    - int ClassName::myFunction(char *s, string name, int i) {
            // C++ code to implement the function goes here
      }

For us, we will put C++ class definitions (mostly) in .h header files, and C++ class implementations in .cpp source files.

To get the program to build properly, we need to place #includes and “using namespace std;” lines in appropriate places. In C++ (and C), a #include line is equivalent to placing all of the contents of the named file at the spot that the #include appears. In practice, we use this facility to organize code, to separate interfaces from (hidden) implementations, and to facilitate separate compilation and program building. Since lab1.cpp and Hello.cpp both use the Hello C++ class, both need its definition at the top of the file. So add the following line near the top of both lab1.cpp and Hello.cpp:

- #include “Hello.h”

You will also need to update the makefile to get the executable to build from more than one source file.

- Alter the makefile to build an additional .o file (Hello.o from Hello.cpp), and to link (using g++) that .o with lab1.o to create lab1. See if you can alter the makefile to build lab1 successfully from the three files. Receiving “feedback” from make and g++ that you have not done so correctly (yet) is part of the point of this exercise. Try to interpret and understand the compiler and make errors that you may encounter.

Part D : Code Organization and Submission

- Required code organization:
  - lab1.cpp
  - Hello.cpp
  - Hello.h
  - makefile
  - readme
- Create a readme file following format provided, and add it to your project folder
- While inside your lab 1 folder, create a zip archive with the following command
  - zip -r lab1 *
    - This creates an archive of all file and folders in the current directory called lab1.zip
    - **Do not zip the folder itself, only the files required for the lab**
- Upload the archive to Mimir under 'Lab 1'

Expected Interface and Test Output

- **Test Commands**
  - ./lab1
    - *Expected Test Output:*
      - Hello World!
        C++ Data Structures!
        Hello from the Hello Class!

Grading Guidelines

- **Part A:**
  - Compiles and outputs hello world when run (1 points)
- **Part B:**
  - Compiles with makefile (1 point)
- **Part C:**
  - Hello class with a method that prints implemented as described (1 point)
  - Implementation (.cpp) properly separated from interface (.h) as described in the lab. (1 point)
  - Uses separate compilation in the makefile to compile (2 point)
- **Part D:**
  - Follows requested project structure (1 point)

    - Contains written Readme (1 point)
    - Follows formatting guidelines (1 point)
    - Clean submission does not include .o files or binary (1 point)

Formatting Guidelines

- Stores all values in a named variable.
    - No Magic Numbers.
- Uses indentation to identify code blocks.
    - Every Code block should be indented from it's parent block to identify scope.
- No single letter or non-descriptive variable names
    - The only exception to this rule is 'i' in a for loop
- Separates code blocks and logical sections with whitespace
    - Optimize your code for the reader, not the writer
- Output is formatted with an explanation of the output values
    - Format your output so that someone who does not know what the program is supposed to do would know what the output meant
- Each method is preceded by a comment explaining what the method does
- Each significant code block is preceded by a comment explaining what the code block does.
    - A significant code block is more than 3 lines performing a single logical operation
- CONSTANTS are in all caps
- Only data types start with a capital letter
    - Classes, Enums, Structs, etc.
- Do not use the 'using namespace' declaration in a header (.h) file
- In general we will follow the Google C++ style guidelines. If you want more info, you can view them here: https://google.github.io/styleguide/cppguide.html