Coding Assignment 0
Getting to Know the Linux C++ Development Environment
CS-240: Data Structures
Assigned Friday August 25, 2017
See MyCourses Blackboard for Due Date and Time

Goal

The goal of Coding Assignment 0 (CA0) is to familiarize you with C++ program development, using the CS Department's linux computer lab and program development environment. CA0 also defines the process by which you will submit this program and all future CS 240 coding assignments. In particular, you will learn how to do the following:

- Log into a linux machine and run several useful linux commands
- Use the linux command named "`man`", which allows you to learn how to use other linux commands
- Create and navigate linux directories
- Create and edit linux files
- Compile, link, and run C++ programs using the g++ compiler from within a linux "makefile"
- Identify and edit a C++ program
- "Tar" and "zip" a set of files into a single file with an appropriate name and format for submission
- Submit a 240 assignment into Blackboard

You may refer back to the latter sections of this assignment when submitting all future assignments, as the process and naming conventions will be the same for all assignments.

Learning about Linux commands, and Creating and Editing a Unix File

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", which on the surface looks like the Windows command environment. We will call this, inter-changeably, a *terminal* or linux *shell*. The shell will accept and execute linux commands, of which there are hundreds. We'll learn a few for this assignment, 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`.

Explore the man pages for the following command (listed below). Using your own words, you will write definitions into a linux file, with some information about five of the commands. For each command, your description should not exceed three lines. Save all the descriptions in a text file, and name the file *LastName_FirstName_userid_Definitions.txt*, where “LastName” and “FirstName” are replaced appropriately

for you. For example, if your name is Fritzy McGillicuddy, userid `fmcg33`, your file should be named *McGillicuddy_Fritzy_fmcg33_Definitions.txt*.

Pick five commands from the list below, and:

- If you had no idea what the command did, provide a very high level description of the command.
- If you already knew generally what the command did, explore further and include one new aspect of the command that you just learned from the man page. (Perhaps there is a command line option that you did not know about?)
- If it turns out you already knew everything about 5 or more of the commands on this list, find others that you are not as familiar with, and write something about those.

The only requirement is that you turn in short blurbs about any five linux commands, comprising information that you did not know before doing this assignment.

- `man`
- `pwd`
- `mkdir`
- `rmdir`
- `cd`
- `ls`
- `find`
- `mv`
- `more`
- `make`

To create a text 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 "`pico`". You may use whatever editor you prefer, but if you are learning one for this class, I don't recommend `vi` or `emacs` (yet). Pico 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_fmcg33_Definitions.txt
```

(`McGillicuddy_Fritzy_fmcg33_Definitions.txt` can be replaced with the name of any file you wish to edit or create, including C++ programs in "`.cpp`" and "`.h`" files.)

If there is no file currently named `McGillicuddy_Fritzy_fmcg33_Definitions.txt`, 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_fmcg33_Definitions.txt`"). 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 `vim` or `emacs`) 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 Blackboard "SYLLABUS & COURSE INFO" area 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. *But please let us know if you need help, and we will be happy to provide it, especially early in the semester!*

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 the permissions of this directory so that no other user can read the files. Use the following command to achieve this:

```
:~> chmod 700 cs240
```

(Later, read the linux man pages for `chmod` to learn about "permission bits" and how they define access to unix files.)

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 *CA0username*, where the "username" part matches the user name you used to log in. (For example, if your username is *fmcg33* (which would be quite a coincidence), you should now have a directory named *CA0fmcg33*. "`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, learned how to create and navigate directories, and learned to create and edit a linux file. In the next few sections, you will edit, compile, and run a C++ program within linux.

Create a file within the CA0 directory you named with your linux username; name the file `Hello.cpp`, and have your file contents match the contents of **Hello.cpp**, which can be found in Blackboard under Coding Assignment 0. You can do this in one of three different ways:

(i) by downloading our `Hello.cpp` file directly, using your browser,
(ii) by creating a new empty file named `Hello.cpp` and cutting and pasting the contents of our `Hello.cpp` into your `Hello.cpp`, or
(iii) by reading and typing the C++ code into a new file.

Option (iii) is not recommended because it will take too long. Option (ii) may cause a strange problem of embedding an extra unrecognizable character inside the file... the symptom of this is that your program will not compile properly when you type "`make`" (see below), or `g++` (an alternate way of building your program).

Copy the `makefile` found in Blackboard into your current working directory for CodingAssignment1. Name the file "makefile" (no file extension).

A `makefile` contains a set of instructions that tells linux how to build the program contained (often) within the current directory. In this case, our `makefile` contains a very simple set of instructions to build a program named `Hello.exe` from a single C++ file named `Hello.cpp` and contained in the current working directory (the directory that contains the `makefile`). Observe the `makefile` and notice that it contains two kinds of lines: some begin with "#" and some do not. The lines that begin with # are comments, and are ignored by the linux `make` utility. The other lines specify the rules for building the program.

To compile the C++ source file, execute the following command:

```
:~> make all
```

This command produces an executable file named "`Hello.exe`" 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 "`make all`," which in our case is contained on the 2nd line of the `makefile`. This rule indicates that "`make all`" should follow the rule for "`Hello`", contained below it. The `Hello` rule, in turn, contains the following two lines:

```
Hello: Hello.o
     g++ Hello.o -o Hello.exe
```

Importantly, there is a `[tab]` character (not a single space) between the "`.`" and the "`Hello.o`", and another one before the `g++.` These two lines indicate that the `Hello` target *depends on* the `Hello.o` file. This causes the `Hello.o` rule (also defined within the `makefile`) to be invoked. The `Hello.o` rule depends on `Hello.cpp`. So if `Hello.cpp` has been updated recently (i.e. since the time that `Hello.o` was created), then the `make` utility will re-run the command that follows (in this case "`g++ -c Hello.cpp`"). This line ("`g++ -c Hello.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 `Hello.cpp`. The command's output is a file named `Hello.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 "`Hello`" target. `g++` also does this linking, this time via the following command:

```
g++ Hello.o -o Hello.exe
```

To repeat (in an attempt to be as clear as I 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`.)

So all of the steps in the paragraph above happened "behind the scenes" when you typed "`make all`." To verify that the two new files (`Hello.o` and `Hello.exe`) were created, run the following command:

```
:~> ls
```

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

The final thing to notice in the makefile is the rule for "make clean," which runs the following command:

```
:~> rm -f *.o Hello.exe
```

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 "Hello.exe"). 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
```

(Just running "`make`," instead of "`make all`", also works.) Run `make` again to observe that the object code and 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:

```
:~> ./Hello.exe
```

(You may also be able to run it by simply typing "`Hello.exe`", without the "`./`" part, which tells the shell to find the program in the current working directory. 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 `Hello.cpp` file to see the program that produced it. To look in the `Hello.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 `Hello.cpp` file and alter the code to have it print "Hello C++ Data Structures, from <Your Name>!!", where <Your Name> is replaced by your name. Immediately upon doing so, save the file and try running the program again:

```
:~> ./Hello.exe
```

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

Changing File Names

The `makefile` is specifically designed to compile a program contained in a file called `Hello.cpp`. Let's now change the name of that file, and update the program to build the newly named file. Before doing so, clean out the generated object code and executable code from your directory:

```
:~> make clean
```

(Notice that the `makefile` we gave you contains a rule for "`make clean`"; you should always include and support that rule in your makefiles, but be careful that you "clean up" only the right files (those that are regenerated by the compiler and linker. A `makefile` that removes your source code is not advised!)

Now rename `Hello.cpp` to `LastName_FirstName_userid_Hello.cpp`, where "`LastName`", "`FirstName`", and `userid` are replaced by your names and userid. For example,

```
mv Hello.cpp McGillicuddy_Fritzy_fmcg33_Hello.cpp
```

Now try building your program by typing "`make all`". This now fails because `make` can no longer find a file named `Hello.cpp` in the current directory. So change the `makefile` to instead look for your new C++ file. You can do this by changing all occurrences of "`Hello`" in the file, with the string containing your name (e.g. `McGillicuddy_Fritzy_fmcg33_Hello`). Make sure the file names in the `makefile` match exactly the ones in the linux directory.

Rebuild the program after making the changes to the `makefile`, verify that the newly named object code and executables are created properly, and run your new executable with the new name.

Separating C++ Code into Multiple Source Files

Next, please separate and move the C++ class named `Hello_Class`, which currently resides in `McGillicuddy_Fritzy_fmcg33_Hello.cpp`, into two different new source files, called `Hello_Class.cpp` and `Hello_Class.h`. Leave the `main()` function in `McGillicuddy_Fritzy_fmcg33_Hello.cpp`. To do this, you will need to separate the *class definition* from the *class implementation*, by moving some code around and changing some syntax. You will also need to update the `makefile` to get the executable to build from more than one source file. Notice that the `Hello_Class` 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*.

Restructure the code in `McGillicuddy_Fritzy_fmcg33_Hello.cpp` into `Hello_Class.cpp` and `Hello_Class.h` (and leave some code in `McGillicuddy_Fritzy_fmcg33_Hello.cpp`), as described above. That is, place the class definition in `Hello_Class.h`, place the member function implementations in `Hello_Class.cpp`, and leave `main()` in `McGillicuddy_Fritzy_fmcg33_Hello.cpp`.

To get the restructured program to build properly, we need to place the `#include` and "`using namespace std;`" lines in appropriate places, and add a new `#include` for `Hello_Class.h`. 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 `McGillicuddy_Fritzy_fmcg33_Hello.cpp` and `Hello_Class.cpp` both use the `Hello_Class` C++ class, both need its definition at the top of the file. So add the following line near the top of both `McGillicuddy_Fritzy_fmcg33_Hello.cpp` and `Hello_Class.cpp`:

```
#include "Hello_Class.h"
```

Next, you need to alter the `makefile` to build an additional .o file (`Hello_Class.o`, from `Hello_Class.cpp`), and to link (using `g++`) that `.o` with `McGillicuddy_Fritzy_fmcg33_Hello.o` to create `McGillicuddy_Fritzy_fmcg33_Hello.exe`. See if you can alter the `makefile` to build `McGillicuddy_Fritzy_fmcg33_Hello.exe` 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.

Packaging Multiple Files Into One "tar ball"

The directory named with your user name should now contain eight different files:

1. Two `.cpp` files with C++ code in them
2. A `makefile`, containing rules for building your code
3. Two `.o` files with C++ object code in them
4. One `.h` header file (`Hello_Class.h`) that contains the `Hello_Class` class definition
5. A `.exe` file with executable code in it
6. A `.txt` file with your definitions for the handful of Unix commands (once you have gone back and done these)

When you submit this program, and all future programs, *we do not want the object code and executable code!* It is too big, and if you submit a `makefile`, we can generate the object code and executable code ourselves anyway, by simply running "`make`". You must submit code that compiles, links, *and* runs. Therefore, to prepare to submit your code, run:

```
:~> make clean
```

Now, rather than submitting the five remaining files separately, we would like you to submit them together in one file. Linux contains two complementary command utilities for doing so, named "`tar`" and "`gzip`". Move "up" one directory by running:

```
:~> cd ..
```

Now create a tar file by running the following command (using your own last name, first name, and userid, obviously):

```
:~> tar -cvvf McGillicuddy_Fritzy_fmcg33_CA0.tar CA0fmcg33
```

Observe that a new `.tar` file has been created, by running the linux `ls` command.

Next, "compress" this file into a gnu "zipped" file by running:

```
:~> gzip McGillicuddy_Fritzy_fmcg33_CA0.tar
```

Verify that this creates a file named ***`McGillicuddy_Fritzy_fmcg33_CA0.tar.gz`***. This is the file that you will submit to us.

You can verify that your code has been properly zipped and tarred by creating a new directory somewhere, using "`cd`" to move into it, and by running the inverse of the "`gzip`" and "`tar`" commands (which are "`gunzip`" and "`tar`" with different command line arguments, respectively). In particular, the following commands will help verify that your submitted code will work properly for us:

```
:~> mkdir CA0TestSubmit
:~> cp McGillicuddy_Fritzy_fmcg33_CA0.tar.gz CA0TestSubmit
:~> cd CA0TestSubmit
:~> gunzip McGillicuddy_Fritzy_fmcg33_CA0.tar.gz
:~> ls
:~> tar -xovf McGillicuddy_Fritzy_fmcg33_CA0.tar
:~> cd CA0fmcg33
:~> make
:~> ls
:~> ./McGillicuddy_Fritzy_fmcg33_Hello.exe
```

CS240 Submission Conventions

In summary, please follow these rules when submitting this and all future Coding Assignments and Assignments:

- Confirm that typing the command "`make all`" and just "`make`" in the appropriate (top level) coding assignment directory correctly builds the executable that you want us to run and test
- Confirm that all files, including and especially the executable file, are named (perhaps by "`make`") exactly as specified in the coding assignment
- Please place your code in a directory named **`Lastname_Firstname_userid_CA0`**. (For future coding assignments, please substitute the appropriate assignment number.)
- Remove all object code, executable code, and other extraneous files from your directory (perhaps by using "`make clean`" to remove some of them). You may leave automatically generated backup source code files in your submission. I do not want you to get in the habit of cleaning those out… they could

come in handy!

- Use `tar` and `gzip` to pack your code into a file named `Lastname_Firstname_CA<N>.tar.gz`, where <N> is the assignment number and the file name reflects your own name.

It is very important that you follow these rules closely, because we will have many submissions to track and manage. Therefore, we will deduct points if you do not adhere to the naming and submission conventions precisely.

Submitting 240 Coding Assignments

Submit Coding Assignments (and Homeworks) in the appropriate Content area of Blackboard. Let us know if you need help with this procedure.