Lab 2 – Building on Linux

Assignment Details


Background

This assignment should introduce the basic development tools on Linux. This assumes you have completed the first practicum and are familiar with it.

The Practicum

First things first, let’s check out a new copy of the source control tree – to keep from cluttering up your multilab account, we’ll try to reuse this one from here on out, but just to make it simpler, let’s check out a new copy.

Refer to the first practicum document if you need; but after finishing the checkout, you should see something like this:
We’ll be using “main_copy” as the standard name for the current source control checkout from here on out. If you have questions about what the above-listed `svn` command means, please ask!

As you can see, the checkout includes directories for the first practicum (la1), Programming Assignment 1 (pa1), and the current practicum (la2). These are all now created on your multilab account as a result of the `svn` checkout command.

You might have a class ready from Programming Assignment 1 Part 1. If not, put together a quick skeleton of a class – something that should compile, at least (There are example files posted on the course website that can be used if you don’t have any prepared¹).

So, assuming you’ve now got these files on your local machine, let’s use Filezilla to move them to where we’re working (the la2 directory above):

---
¹ Please, though, don’t wait until the last minute for PA1 Part 1, as it does present a good deal of sheer typing to get it finished...
Navigate to where on the local machine you have the files, navigate to the la2 directory on your multilab account, and drag the files from the left (local) panel to the right (remote) panel.

Now, back to the multilab login – we’ll need to navigate to the directory; you can do this in two different ways:

```
rose:~> cd main_copy
rose:~/.main_copy> cd CS216
rose:~/.main_copy/CS216> cd ddbstudent
rose:~/.main_copy/CS216/ddbstudent> cd la2
rose:~/.main_copy/CS216/ddbstudent/la2> ls
./ Item.cpp Item.h .svn/
rose:~/.main_copy/CS216/ddbstudent/la2>
```

Or:
Both of which do the same thing – the important thing to take home is that navigating through the directory structure via the cd command on the multilab machines is exactly analogous to doing so through the graphical user interface from Windows – you can see directories (with the ls command), and change them with the cd command.

Now that we have the files copied over and our multilab login set to that folder, check to make sure that the files show up with the ls command.

This practicum we’ll be using the g++ command (short for “GNU C++”) to compile some stuff.

To start with, run g++ on one of the .cpp files you uploaded; if it’s a file from either PA1 or the samples from the course website, you should see something like this:
Which is... an awful lot of stuff that isn’t really necessary. But the core bit is right next to the end: “undefined reference to ‘main’”. Recall from CS215 (or earlier time spent with C++) that for a C or C++ program to function, it must have a main function, and the PA1 files won’t have one.

So, let’s write a new file and put a main function in it. We’ll start with a simple Linux text editor, pico (which actually loads up GNU nano, but still):

```
(.text+0x20): undefined reference to `main'
collect2: ld returned 1 exit status
rose:~/main_copy/CS216/dbbstudent/la2>
rose:~/main_copy/CS216/dbbstudent/la2> pico main.cpp
```

Which, when executed, gives something that looks like this:

![GNU nano 2.2.6](image)

And other than the lack of mouse selection, it works a lot like notepad on a Windows machine; cursor keys move, and you can type in text. So let’s put together a simple “Hello, world”:
Hit control-X to exit, tell it you want to save the file, and…

```
#include <iostream>
using namespace std;

int main(int argc, char * argv[])
{
    cout << "Hello, World" << endl;
    return 0;
}
```

So now we have a new .cpp file which should actually compile, as it has a main function. To do so, we’ll call g++ on it:

```
rose:~/.main_copy/CS216/dbbstudent/la2> g++ main.cpp
rose:~/.main_copy/CS216/dbbstudent/la2>
```

And… nothing? In the Unix world, it’s common for programs to indicate success by generating no output what-so-ever. But if we use the ls command to check the directory…

```
rose:~/.main_copy/CS216/dbbstudent/la2> ls
./  ../  a.out*  Item.cpp  Item.h  main.cpp  .svn/
rose:~/.main_copy/CS216/dbbstudent/la2>
```
We see we have a new file – a.out. This is the default name for an executable generated by 
g++. And so, now that we have an executable, the next step is to run it. In Linux, the typical method for running an executable from the current directory is typing “./” before the name\(^2\), as shown:

```
rose:~/main_copy/CS216/dbbstudent/la2> . ./a.out
Hello, World
rose:~/main_copy/CS216/dbbstudent/la2> _
```

Ok! Program written, compiled, and executed! Now that we’re caught up to week 2 of CS215, let’s move on.

So, let’s go back to main.cpp – open it back up in pico (use `pico main.cpp` at the command line), and let’s add a reference back to the files we copied over:

```
#include <iostream>
#include "Item.h" // Change this if you have a different file
using namespace std;

int main(int argc, char * argv[])
{
    // You'll need to interact with the other 
    // files in *some* way
    Item item;
    item.setName("Wombat");
    cout << "The item is a " << item.getName() << "." << endl;
    return 0;
}
```

Note that to get full credit your code here will need to interact with the files you copied over.

\(^2\) The reason for this is a bit outside the scope of the class, but is, primarily, for security reasons – to prevent someone from accidentally running a malicious program named “cd” or “ls” or whatnot placed in a directory by an attacker.
And, next we compile it…

```
rose:~/main_copy/CS216/dbbstudent/la2> g++ main.cpp Item.cpp
```

Aaaand we have a problem.

What we see here is a series of “undefined reference” errors; the problem is while we compiled main.cpp, the actual logic for the Item class (in this example) is stored in Item.cpp – and we didn’t tell g++ to compile that file. Note that we won’t need to compile the header file – that gets utilized by g++ as a result of the #include directive in main.cpp.

So, to fix it, we can pass both files as arguments to g++:

```
rose:~/main_copy/CS216/dbbstudent/la2> ls
. ./ a.out* Item.cpp Item.h main.cpp .svn/
rose:~/main_copy/CS216/dbbstudent/la2> ./a.out
The item is a Wombat.
```

Or, we can do it a bit simpler, using a wildcard:

```
rose:~/main_copy/CS216/dbbstudent/la2> g++ *.cpp
```

Aand… we’re done.

We’ll talk more in the near future about ways to automate g++ and do more with it (and the difference between the compilation and linking step), but all of that will come up later in the class – for today you just need to be comfortable with navigating around Linux, and compiling and running programs from the command line.

**Submissions**

For full credit for this assignment, you will need to have the following checked in to the la2 directory in source control:
• At least two classes defined in a .h file and implemented in a .cpp file (and you can even use the barebones ones given on the web site)
• A main.cpp that calls functionality compiled in another .cpp file
• A screenshot showing you compiling the two .cpp files and running the output
• Remember that to be submitted, it must be checked in to the source control system – you must use the “svn add” command to add new files, and then “svn commit” to actually send them to the source control server – if your commit command did not execute correctly, the assignment is not submitted!

---

3 Yes, this is somewhat redundant from last week, but I want everyone to be very used to moving files to the multilab.