Lab 1: C++ Development on the Linux Command Line

Welcome

Hey there! Welcome to your CSCI 1730 lab. For each lab this semester, you will have a document just like this to guide you. This material (as well as in-lab instruction) is intended to supplement lecture instruction by focusing on the hands-on aspect of programming. What better way to learn than by doing! Plus, you'll get individual help if when you run into problems.

Most labs will involve some reading, some computer activity, and a deliverable to turn in for a grade. We'll try to keep the labs interesting, informative, and relevant for all skill levels. The best way to work is to read straight through, working on your computer when asked. You may not finish by the end of the class period, but you should be able to work from anywhere with an Internet connection.

Introduction to This Lab

Linux was made by developers for developers. It's an essential tool to know how to use, and can be both incredibly simple and powerful at the same time. In this lab, we will focus on the essentials required to get you up-to-speed for this class. You will need to understand what's going on to work on your projects later on.

Goals

- Get connected to nike and your VM
- Learn the basic commands required to navigate the environment
- Compile and run your first C++ program
- Explore the options for text editors

Get Connected

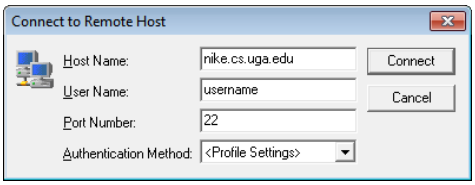
In the past, you have probably done all development on a local machine sitting right in front of you. For this class, you will be developing remotely, on a machine provided by the Department of Computer Science. There are many reasons for this. For one, expecting everyone to install the Linux operating system on their computer would be unreasonable. Also, this way, every student has the same environment, tools, and computing power available to them. The

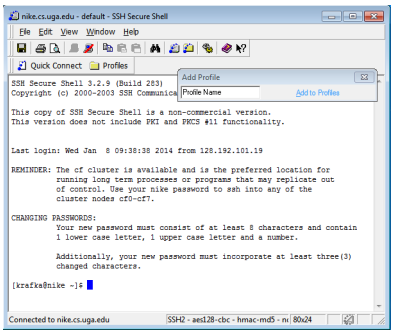
downside to this is that you will need an Internet connection to work, but countless real-world systems are administered this way every day.

First, we'll connect to nike. This is a shared server. You may have used it before in CSCI 1302. You will need a username and password set up specifically for this server before you can continue. Questions regarding nike can be directed to support@cs.uga.edu.

> ⓘ **Note:** This document will assume you're using one of the lab computers. You are welcome to use your own computer (which may be good to prepare for working on assignments), but instructions may vary. For more information on setting things up for your own computer (Windows, Mac, and Linux), short video tutorials will be made available on the class website.

1. Open *SSH Secure Shell Client*.
2. Click "Quick Connect."
3. Fill in the "host name" (`nike.cs.uga.edu`) and your nike username, then press "Connect." (Leave the other fields set to their defaults.)
4. Type in your password when prompted.
5. Once you're connected, SSH Secure Shell Client may ask you to create a profile for your connection—feel free to give it a name (e.g. "nike") so you can get logged in faster via the "Profiles" button. Assuming everything went well, you should now see a prompt with your block cursor right after the $ symbol. This means you're ready to start typing commands!





Essential Commands

Now, let's try out some commands. You might as well commit these to memory since you will need to use them on a regular basis. If you know these commands already, we'll try and keep it interesting by throwing in some interesting facts along the way.

6. **`pwd`** stands for "print working directory." This command will display your current location in the filesystem. On the command line, you are in a single directory at any given time, much like an Explorer (Windows) or Finder (Mac) window. Type in `pwd` and press `enter`.

   You should see something like `/home/ugrads/username`. That's the path to your home directory. Each time you log in, you will start here. In fact, most all work you do will be inside your home directory (or a subdirectory

within it), since you have limited permissions elsewhere. Another name for your home directory is simply **~** (the tilde character), which you will see from time to time.

7. **`mkdir`** will allow you to create a directory (called a "folder" in Windows and Mac). This command requires one *argument* after the command: the name for the new directory. Let's create a directory to house our 1730 work. Type the command `mkdir 1730`. Even though nothing was printed out, the "1730" folder was created in your current working directory.

   `rmdir` will remove a directory, as long as it's empty. Keep this in mind, but don't delete your 1730 directory.

8. **`cd`** is short for "change directory." It allows you to change your current working directory.
   a. Go into your "1730" directory by typing `cd 1730`.
   b. Now make a directory called "lab1." It will be created in your 1730 folder.
   c. Go into your "lab1" directory now.
   d. Run `pwd` to see exactly where you are.

Up until now, we've been going down the directory tree. Running `cd ..` will take you up one level. Notice that there's no need to specify the parent directory's name, because there can only be one direct parent! Play around with `cd` until you're comfortable navigating in and out of folders. Use `pwd` along the way to make sure it's working. You can even `cd ..` up to **`/`** (the root).

> ✎ **Tip:** `cd ~` (that's the tilde character again, right above your `tab` key) or just `cd` will take you back to your home directory from anywhere. `cd -` will take you back to the last directory you were in (which is different from `cd ..`, which goes to the parent directory). You can also specify multiple directories at once (e.g. `cd ../../dir1/dir2/` or `cd ~/1730/lab1/`) or absolute paths (e.g. `cd /home/ugrads/username/1730/lab1/`).

9. **`ls`** (as in "list") prints the contents of the current working directory. Use `cd` to return to your home directory and run `ls`. Unless you already have some files on nike, the directory listing should only have your 1730 directory.

   Let's run the `ls` command again with the `-a` *flag*: `ls -a`. We can use flags with many commands in Linux to modify how the command will operate. In this case, the `-a` flag specifies that the ls command should list all files, including hidden ones. In Linux (and other Unix-like systems, such as Mac OS X), hidden files simply have a period at the beginning of

the filename. You can access them like any other file; they are just omitted from the default directory listing.

Many system configuration files are hidden. You may find a few in your home directory; perhaps `.bash_profile`, `.bash_history`, or `.forward`. One thing's for sure, though: every directory has two special hidden directories in it: `.` and `..` (one and two dots). One dot is a link (i.e. shortcut) to the current directory (i.e. every folder has a link to itself). Two dots is a link to the parent directory. You've seen `..` in practice with the `cd` command, but try out the `cd .` command. As you might expect, `cd .` is useless, but the `.` shortcut will come in handy later on.

Now, run `cd /usr/bin/`. This is the directory where many system programs are located. You don't have permission to do much here, but there are a lot of files for us to list. Type `ls`. Notice how the files are displayed in a column format. We can choose to display the files differently if we want. Try using the `-l` flag with `ls` to display the listing in long format. Notice all of the extra information displayed!

What if you wanted to list files in the long format and show hidden files? You have options, actually. You can accomplish it with any of these synonyms:

- `ls -a -l`
- `ls -l -a`
- `ls -al`
- `ls –la`

10. Now that you've got a good idea of how these commands work, let's look at a few more common commands in practice:
    a. `cd` to your 1730 directory (`~/1730/`).
    b. Run `touch test1 test2`. The `touch` command creates an empty file.
    c. Run `ls` and you will notice two files got created! Spaces on the command line indicate different arguments. If we intended to create a single file called "test1 test2" we should have used quotes (e.g. `touch "test1 test2"`) however for this reason, you should avoid using spaces in filenames (underscores are a good alternative).
    d. Move these two new files into the lab1 directory:
        i. `mv test1 lab1/`
        ii. `mv test2 lab1/`
    e. `cd` into the lab1 directory.
    f. Let's make a backup copy of the file "test1": `cp test1 test1.bak`. `ls` to make sure the backup file got created. Notice how `mv` and `cp` use the format `[command] [source] [destination]`.

g. `rm test1` to delete the original file. `ls` to be sure it's gone.
h. There's no Recycle Bin or Trash on the Linux command line where we can recover this file from! Good thing we made a backup. Let's rename the backup to have the original filename. You might guess "rn" would be the rename command, but the `mv` command can accomplish a rename! Run: `mv test1.bak test1`.
i. Don't do this just yet, but when you're done on nike, you can disconnect by doing any of the following:
    - Run `logout`
    - Run `exit`
    - Type `control`+`d` (which indicates "end of file")

If you made it through all of that, nice job! Those are some important essentials to managing files. There are still many commands to learn, but we'll see them as needed.

> ⓘ **Note:** As you work with Linux, you might notice that not all files have extensions. File extensions (e.g. .txt, .mp3, .exe) are particularly useful in graphical environments when you expect to open a file without specifying which program should open it. For example, double clicking a .docx file should open in Word. On the command line, we will specify the program to use before the filename, so the extension is really only useful to help us remember what kind of file it is. One exception to the rule is with executable files: they typically have no extension, and you will run them by just typing the filename. Also, programmers often leave off extensions from common files such as "README" (typically a plain text file). In any case, a file is file: just a line of ones and zeros. The program you use to open it depends on how it will be interpreted.

> ✎ **Tip:** Use the `tab` key to autocomplete filenames. Just type the first few letters and `tab` will type the rest. If there's an ambiguity, you will hear the system bell; press `tab` again twice to see your options, then type more of the one you want before trying again.

Your First C++ Program

We haven't talked about how to use any command line text editors yet (and we don't know C++), so we'll be downloading pre-existing source code to start out.

11. Make sure your current working directory is `~/1730/lab1/`.

12. Run the following command to download your first C++ source file to the current working directory:

```
wget http://cobweb.cs.uga.edu/~eileen/1730/Labs/Lab1/hello.cpp
```

13. Examine the source code by running `cat hello.cpp`. **`cat`** is short for "catenate" (a synonym for "concatenate"), because you can provide multiple filenames as arguments to display the contents of multiple files, back-to-back. Most often, we'll just use `cat` to see what's inside a single file. You can probably guess what most lines in the source code do, but we'll worry about that another day.

14. Let's compile it! Run `g++ -o hello hello.cpp`. Assuming everything went right, you shouldn't see any output, but if you run `ls`, you should see a new file called `hello`.

    What does that command mean? Well, `g++` is the name of the compiler we're using, `hello.cpp` is our source file, and the `-o hello` argument pair is a flag that indicates the output file (an executable) should be named `hello`. The `-o hello` and `hello.cpp` strings can appear in any order, but the `-o` and `hello` parts must stick together, even though they are technically two arguments. If you don't specify the output filename, you will get a file called `a.out`, which is pretty boring and not very descriptive!

15. Let's run it! Type `./hello`. You should see the output `Hello World!` if everything went right! Congratulations!

    There are a few things worth noticing about the way we ran our executable. First, why did we add `./` to the beginning? Recall that `.` is a shortcut to the current directory. It's correct to indicate that the `hello` command is located in the current directory, but why is it necessary? Usually, the current directory is assumed, but when you run commands, the system will not look in the current directory. This is for security and sanity reasons. Imagine what would happen if you created an executable file called "rm" (the same the remove command). You might have trouble removing it if typing `rm` only called your local rm executable!

    Also, compare this to the way you run Java programs on the command line. In Java, this exercise might look be run as `java Hello`. Because Java runs in a virtual machine, `java` is the executable being run and your compiled code is input into the `java` command. In C++ and C, we will be creating executables that can be run directly on the machine.

Nice work! You are now officially a C++ programmer, even if you didn't write the code. :-)

Using Command-Line Text Editors

Perhaps the most important tool you will use on the Linux command line is a text editor. Considering source code is text and the command line environment is text-based, you might be spending considerable time with the text editor. Fortunately, you have some very powerful tools at your disposal.

Specifically, you have two editors to choose from: **`vi`** (pronounced as two distinct letters, "v" and "i") and **`Emacs`** (pronounced as the letter "e" followed by "macs"). Historically, programmers have fought about which editor is better (typically all in good fun), so choosing a side is an important part of your career as a computer scientist! We'll take a look at the basics of both editors, but you are welcome to use either one. You're likely to find both editors on any given Linux system.

Be warned: Both editors have a steep learning curve, but taking the time to learn useful editing techniques now will save you time later. There are more basic editors available, notably `nano` and `pico`, but they are far from ideal for writing code. Both `vi` and `Emacs` will color your code, if the file is named with an appropriate extension.

vi introduction

Technically, nike and other modern Linux machines will launch **`Vim`** when you ask for `vi`. `Vim` stands for "Vi IMproved" and is pronounced like "rim." We'll refer to it as `vi` for historical reasons, and since the basic functionalities are the same. "vi" is actually short for "visual," because the editor is more visually oriented than its predecessors.

16. Make sure your current working directory is the lab1 directory you created earlier.

17. Run the command `vi editor_test_vi.txt` (or change the filename as desired). This will create a new file. If a file had already existed by that name, it would have been opened in `vi`. **Don't start typing just yet!**

18. When you open up `vi`, you are in *command mode*. Any letter you type likely has an associated command. Since we want to start adding text, we need to switch to the other mode: *insert mode*. To do this, type `i` (just the lowercase letter). You will see it says `-- INSERT --` at the bottom to indicate that you are in insert mode.

19. Type something! Add at least five lines of text with a few words on each line. You can use the arrow keys to move your cursor around if needed.

20. You can always get back to command mode by pressing `esc`. Let's do this now. Now we'll try a few commands:

a. Move your cursor down and up by pressing `j` and `k`, respectively. (Arrow keys work too, but many `vi` users prefer to keep their fingers on the home row.)
b. Move your cursor left and right by pressing `h` and `l` respectively. This is understandably uncomfortable; however, with time it can become muscle memory. These (and other) `vi` key bindings show up other places as well, including Gmail to move up and down between conversations (if enabled), and Facebook to scroll up and down the News Feed.
c. Jump to the top line of your file by typing `g` `g`.
d. Delete a line by putting your cursor anywhere on it and pressing `d` `d`.
e. Repeat the last command by pressing `.`.
f. Press `y` `y` to *yank* (i.e. copy) an entire line, then press `p` to paste it.
g. Press `3` `p` to paste it three more times!
h. Move your cursor to the beginning of a word and press `d` `w`. This will delete the word.

21. Enough messing around! Let's write (i.e. save) our document and quit `vi` by typing `:wq` and pressing `enter`.

Emacs Introduction

`Emacs` does not use the bimodal system that `vi` does, but rather relies on the use of modifier keys. The notation used is `C` for the `control` key and `M` for the "meta" key, which is either `alt` or `esc`. For example, `C-x C-c` means to press `control`+`x` then `control`+`c`.

22. Make sure you are still in the lab1 directory.

23. Run `emacs editor_test_emacs.txt` to create a new file by that name, just like we did with `vi`. You can start typing in text right away. Add about five lines of text.

24. Like `vi`, you can navigate with arrow keys alone, but there alternatives. `C-f` and `C-b` move the cursor forward and backward (one character at a time), respectively. `M-f` and `M-b` forward and backward a word at a time. `C-n` and `C-p` move to the next and previous lines. Try this out to get a feel for it.

25. Use `C-a` and `C-e` to jump to the beginning and end of the current line. These key bindings also show up from time to time in other places. For example, Mac OS X implements some of these cursor movement key bindings in any standard text field!

26. Also, like `vi`, you can repeat commands. Use `C-u #` before a command to repeat it # times. For example, `C-u 5 C-f` will advance the cursor five characters.

27. Now, let's save our work and exit. `C-x C-s` will save and `C-x C-c` will close Emacs. This may seem overwhelming but again, with practice, your fingers will fly!

> ✎ **Tip:** Do some exploring on your own! There is a plethora of features for both `vi` and `Emacs` you will find useful. There are so many timesavers, and even a few time wasters (e.g. try out `M-x pong` `enter` in Emacs!)

Conclusion

Now, you are familiar with getting connected to nike, navigating the Linux filesystem, compiling a C++ program, and using two advanced command-line text editors. Sure, we've only seen the basics, but future labs will introduce helpful tools as necessary. You will become more comfortable with everything in this lab as you begin to work on projects for this class. Improving your skills in a text editor will not likely be touched on again, so make it a point to do some exploring!

You're well on your way to a very successful, interesting, and practical semester!