An Introduction to the UNIX Command Line

Created by Brandon Tarquinio, Sarah Gunderson, and Robin Cosbey

Contents

Chapter 1

The Basics

1.1 Introduction

In this workshop we will run through the basics of using *bash* or some other Unix-like command line environment. *bash* is the most popular of the different *shells* (it is then default on most Linux distros and OSX) but most things we discuss will be universal to all shells for Unix-like computers.[1] We will run through enough of the basics that the blinking cursor of a shell is no longer daunting to you. This will include how to manipulate files and directories, use built-in manual pages, and how to run your programs for your classes. Let's start by opening the terminal:

```
ctrl + alt + c
```

1.2 Running a Command

The general form of any command, for the most part, is as follows:

```
commandname -arg1 -arg2 ... --longerargs ... input1 input2 ...
```

Note that *arg* is a common shortening of argument and that *longerargs* is preceeded by two dashes while arg1 and arg2 are preceeded by only one.

The "..." above are used to denote that we can have any number of args of the first type followed by any number of args of the other type followed by any number of input.[2] Lets look at an example:

```
head -v --lines=20 myfile
```

We will go over **head**, along with many other commands, later but for now we just want to understand the general form of commands.

Also note that the double dashed longerargs arguments will only work on Linux and cannot be used on Mac OSX for example. Generally there is an equivalent short arg version. For example we can run the following on Linux or OSX:

[1] For example Linux, OSX, FreeBSD. Also Cygwin provides a Unix-like environment for Windows computers.

[2] In actuality every command has a finite number of arguments it understands and you normally use a small subset of those. There might also be a limit on the number of input.

```
head -v -n 20 myfile
```

Which is exactly the same as our previous example.

1.3 Directories and Moving Around

1.3.1 Directory trees and pwd

We will start by learning some commands to move around the directory tree. It is called a directory tree because it can be visualized as follows:

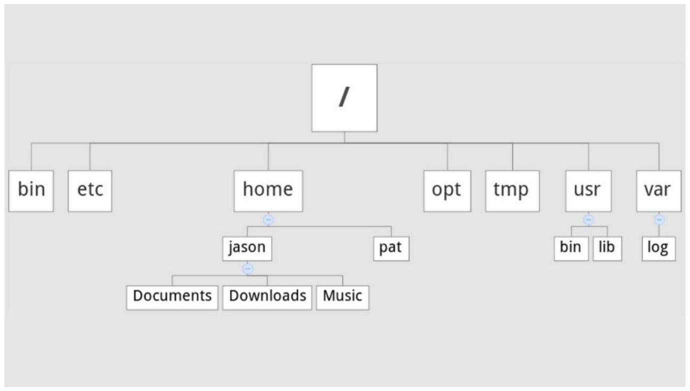

Figure 1.1: An Example Linux Directory Tree.

Each of the boxes in Figure 1 are directories and we call the directory "/" *root* since it is the root of our tree. In a command line enter:

```
pwd
```

This will **p**rint your current **w**orking **d**irectory. For example if we were in the directory *jason* and typed **pwd** we would get the following:

```
/home/jason
```

Note that your *prompt* (the thing that is printed after every command which allows you to enter the next) might be set up to display this information. The prompt is something you can change to your liking to include, or not include, a bunch of information but this is beyond the scope of this workshop.

1.3.2 Listing the contents of a Directory

To **lis**t all the files in your current working directory enter:

```
ls
```

In our example we get the following printed to the terminal:

```
Documents Downloads Music
```

Most shell environments can be set up (and yours might already be) so that directories are printed with a different colour than files and soft links. We will be discussing the difference between files, soft links, and directories shortly. We can also specify to **ls** what directory we want to list the contents of with the following generic form:

```
ls dirname
```

For example:

```
ls /home
```

will display the following:

```
jason pat
```

1.3.3 Changing your Current Working Directory

We have talked a lot about current directories changing but have yet to figure out how to do so. We can **c**hange the **d**irectory with **cd** which has the following form:

```
cd dirToChangeTo
```

For example we can run:

```
cd /
```

And then:

```
pwd
```

and we will see:

```
/
```

To change back to our previous directory with this shortcut:

```
cd -
```

To change to our home directory regardless of where we are we can simply type:

```
cd
```

Thus the default to **cd** is to change to the users home directory.

Exercise 1.1. *Use **cd**, **pwd**, **ls** to explore the directory tree a bit. Becoming familiar and quick with this is key to become efficient with the command line since **cd** and **ls** are easily the two most used commands.*

1.3.4 Relative vs Absolute Path Names

We need to learn one more important concept about UNIX files and directories before moving on to learning a bunch of commands and how we can use them to help us program. Suppose we are in the directory /home in our directory tree figure and we wanted to list the contents of the jason directory. We can do this in two ways:

```
ls jason
```

or

```
ls /home/jason
```

In the later we used the **absolute path name** which is the name of the file or directory prefixed by the path from "/" to it. When we use absolute path names it no longer matters what are current working directory is e.g. we could have run the second command from any directory. In contrast, the first command will only list the contents of /home/jason if are current working directory is /home. This is because jason is a *relative path name* since we assume the path begins in our current working directory. relative path names allows the user to remember and type less but require careful consideration of what your working directory currently is.

1.3.5 Special Directory Entries and Hidden Files

First we will define what the *Home* directory is

- Home: Every user has a home directory. When you log onto a terminal you will start in your home directory. Think of this as the root of the tree which is all of your personal files.

Exercise 1.2. *Lets explore the concept of home by running the following commands:*

```
pwd
```

```
echo $HOME
```

```
echo ~
```

What did you find? Note that "∼", the tilde symbol which is located to the left of the 1 on your keyboard, and "$HOME" are shortcuts to your home directory. Remember "∼" since it will be used a lot.

Hidden files and directories are generally configuration files that are not shown unless you specify that you would like to see them. These entries are hidden to de-clutter the file system. Now we will review two important hidden directories that are useful shortcuts.

- ".": This represents the current directory and is an actual entry in every directory.
- "..": This represents the parent of the current directory and is also an actual entry in the current directory.

Exercise 1.3. *I said before that "." and ".." are actual entries in every directory but when we ran* ***ls*** *earlier we did not see them. The -a argument*[3] *for* ***ls*** *will solve this mystery. Enter the following commands:*

```
ls -a
```

```
ls -a .
```

```
ls -a ~
```

```
ls -a $HOME
```

```
ls ..
```

Does the output make sense? As a challenge question what will the following command do? (Try to think it out before running it!):

```
ls ./././
```

1.3.6 Making a Directory

Now that we understand the directory tree and how to move around it we are ready to learn how to add to it. The general form of the command to **m**a**k**e a **dir**ectory is:

```
mkdir newdirname1 newdirname2 ...
```

Exercise 1.4. *Run the following sequence, before the second and third call to* ***ls*** *think about what you would expect to see:*

```
clear
```

```
ls
```

```
mkdir FirstDir SecondDir
```

```
ls
```

```
mkdir FirstDir/Foo FirstDir/Foo2
```

```
ls FirstDir
```

clear *is a useful command that clears the screen.*

Exercise 1.5.

1. *make a directory called ".ghost".*
2. *run:*

   ```
   ls
   ```

   What happened?
3. *Can you think of a command we have learned to be able to see ".ghost"? A directory or file that begins with a "." is said to be a hidden file or directory. They are mostly used for configuration files. Do you see any hidden files in your home directory?*

[3]The a is short for all.

1.3.7 Deleting a Directory

The program to **re**move a **dir**ectory, **rm**, has the following form:

```
rmdir dirname1 dirname2 ...
```

This command will only work if the directories you are trying to delete are **empty**. An empty directory is a directory which only has no other entries but "." and "..". Thus dirname1 is empty if we run:

```
ls dirname1
```

and nothing appears. Note that this is not like windows where there is a recycle bin for files and folders that are deleted; in Unix when you delete something it is gone! In turn you should be very careful when you delete directories and files (we will learn how to delete files and non-empty directories later).

Exercise 1.6. *1. Start by making sure you are in the home directory using* ***pwd*** *and* ***echo*** *∼. Also make sure you still have FirstDir and SecondDir by using* ***ls****.*

2. *Use* ***rmdir*** *to delete SecondDir.*

3. *Use* ***ls*** *to make sure it is gone.*

4. *Make three new directories SecondDir, ThirdDir, and ThirdDir/Foo.*

5. *Delete the three directories you just made. Did you have any issues? Can you delete all three in one command?*

1.4 Basic Shortcuts

1.4.1 clear

Is your terminal covered in lines of code and difficult to determine where one previous command ends and another begins? Typing "clear" into the terminal will wipe away all of the old commands (command shown below).

```
clear
```

It is possible to refer back to those deleted commands by scrolling up in the terminal.

1.4.2 ctrl ($\wedge$)

Sometimes within bash ctrl will be denoted by " $\wedge$ ". Ctrl is included with several commands, one of the more relatable being ctrl-C. This will terminate the process.

1.4.3 tab completion

When we enter tab at the prompt it will try to finish whatever we are currently typing as long as there are no ambiguity. For example if we only had "file1" in our current working directory we could edit it by tying "vim f" and hitting tab. It will fill out the rest of the word so that you now see:

```
vim file1
```

If instead we had both "file1" and "file2" then when we pressed tab it would have only filled in this much:

```
vim file
```

This is because it does not know if you want a 1 or a 2 to follow next. To see all option press tab twice and the list of all options will be displayed on the screen for example typing the following and hitting tab twice:

```
vim f
```

will result in: exampleoutfile1 file2

Exercise 1.7. *One way this can be very helpful is if you forgot a command but remember what it begins in. Enter the following followed by two tabs:*

```
wh
```

What happened? Does anything look familiar? Try a couple other beginning's of commands followed by double tab.

1.5 Editing Files

We should now have a solid understanding of directories but what is a directory without files to store in them? We will now learn a few ways to make files. In Computer Science courses we will use files for lots of reasons:

- Your Program source code: Whether you are taking a class in Python, Java, C, or some other language you will need a file that is your program.
- Programs themselves: In Unix, all of the programs, the ones you write and the ones that are installed, are themselves a file. For example type the following commands:

  ```
  which ls
  ```

  ```
  which python
  ```

  which is a command that tells you where the program is in the computer. It is especially useful in the case of python since most computers have multiple versions of python and which can tell you which python you use when you type python.
- Plain text files: For example, input to or output from your program will be saved in text files.

1.5.1 Nano

nano is a basic command line editor that will come on almost any Unix computer you ever use. It is not the best for writing programs in but it is simple to learn. The following is generic way to make a new file with nano:

```
nano newfilename
```

It will open the editor full screen in your terminal. On the bottom you will see some of the commands you can use where the carrot (this symbol " ˆ ") represents the control key. You can enter text by just typing.

Exercise 1.8. *In this exercise we will make a file with nano.*

1. *If you are not already in FirstDir then change your directory to FirstDir. Remember you can check where you are with* ***pwd*** *and you can use* ***cd*** *to get to FirstDir.*
2. *Now make a file named myfile using the following command:*

   ```
   nano myfile
   ```

3. *Type the following line:*

   ```
   I am making my first file in nano!
   ```

4. *Saving: Notice on the bottom of the screen it says "WriteOut" with a "ˆO" next to it? That is how we will save our file. Save by first hitting Ctrl+o and see that it prompts you for the file name on the bottom of the screen. Now hit enter. We should now see a new message on the bottom of the screen saying "Wrote 1 line".*
5. *Exiting: Now that we have saved some text lets leave by hitting Cntr+x. Now we are back on the command line.*
6. *Use* ***ls*** *to check if our file is in FirstDir.*
7. *Check the contents by using the* ***cat*** *command as follows (we will learn more about reading text files in the next section):*

   ```
   cat myfile
   ```

8. *Edit the file to have the following line added on the next line of the file*

   ```
   I am editing my file to have 2 lines
   ```

   by entering

   ```
   nano myfile
   ```

9. *Now add a third line of whatever you like.*
10. *Now save and exit. Also use* ***cat*** *again to see your changes.*

That is about all you need to know to make some simple files using nano.

1.5.2 Vim

Another editor that will always be present in a Unix environment is the program **vi** or it's big brother **vim** which stands for **vi** **im**proved. **vi** and **vim** are much more powerful than nano but have a much steeper learning curve. In turn we will have a dedicated workshop on how to use them. For now we will give the bare minimum. You can edit a file called "myvimfile", or make one if that file does not already exist, with the following command:

```
vim myvimfile
```

This will open up a full screen editor similar to what happened with nano. You will notice tilde characters along the left side which signify that those lines are not part of your text file. There is also a bar along the bottom that will tell you the name of your file along with it's length (or in the case of a new file it will say "[New File]"). We are currently in a Command Mode which means that what we type will not be entered on the text file but instead give commands to Vim. There are many things we can do in this mode but in this workshop we will only learn how to type. Hit the i key to enter **I**nsert Mode. You can now type exactly like you would in any other text editor.
First hit i and notice the bottom line change to have the following on the left side:

```
– INSERT –
```

This will be there whenever we are in Insert Mode. Enter some text such as the following:

```
Here is a line of text that I entered in Vim
```

To leave Insert Mode hit the Esc key. Notice that the bottom has changed to be blank on the left, this means that we left Insert Mode. Now to save, or **w**rite, our changes and leave, or **q**uit, Vim enter the following:

```
:wq
```

After you hit enter you should be back at the command line with your new file in your current working directory.

Exercise 1.9. *This assignment is too long to do during this workshop but when you get home enter the following to beginning a tutorial of how to use Vim:*

```
vimtutor
```

This tutorial last about 20 minutes and gives you enough to be able to start editing files with Vim using some of vim's power. If you are interested in learning more you should also visit our Vim workshop later this quarter.

1.5.3 Others

There are many more pure command line editors but the above two are the only ones that are both always available and are still useful for general purposes. We can also use the command line to open up one of our favorite gui[4] editor with commands like:

```
emacs myfile
```

[4]gui stands for **G**raphical **U**ser **I**nterface

```
gedit myfile
```

```
atom myfile
```

Note that when you run one of the above commands the will open a gui window and you will no longer be able to use the command line until you close the newly opened window. You can get around this by doing the following:

```
gedit myfile &
```

This will open gedit (or whatever program you choose) in the *background* and allow you to continue using the command line.

1.6 More on Files

1.6.1 Reading Files

There are many ways to read files; we will discuss a few now. For starters you can always use a program you use to editing text files with to read a file (for more on this see the Editing Files section). In general though you should use the right tool for the right job and if you don't want to edit a file you should use a viewer such as **less** or **more**. For example, to open the file *myfile* with **less** type:

```
less myfile
```

This will open the file full screen in the command line and allow you to move up and down the document line by line. You can do this by pressing the up and down arrow keys. For those familar with Vim you can use Vim's way of moving around in less (which is why less is so cool!). Also some terminals are set up so you can use the scroll wheel on your mouse. To use **more** to open the file *myfile* type:

```
more myfile
```

This program is more primitive and is mostly used for reading though large chunks at a time instead of normal viewing. Hit space to move to the next page until you are done or press q to **q**uit at any time.[5]

Some other ways of viewing include **cat**, **head**, and **tail**. cat's main purpose is for con**cat**ination but it can also be used to dump a file to the screen by typing:

```
cat myfile
```

This is great for small files but horrible for larger ones. Head and tail will grab the first few lines or the last few lines, respectively.[6]

Exercise 1.10. *Before moving on take a second to run each of the following lines one by one and think about why each program could come in handy for different situations:*

```
less myfile
```

[5]Pressing q to quit is also used in less.
[6]On my computer this defaults to 10 lines but it can be changed to any number of lines.

```
more myfile
```

```
cat myfile
```

```
head myfile
```

```
tail myfile
```

```
head –lines=2 myfile
```

Note: The above has two dashes before ”lines”.

1.6.2 Copying a File

We can **cop**y a file by **cp** command which has the following generic form:

```
cp sourceFileName distinationFileName
```

For example:

```
cp myfile myfile.backup
```

will make it so the file name *myfile* in my current working directory is copied into a new file called *myfile.backup* which will also be in my current directory.

1.6.3 Moving a File

We can **mov**e a file by the **mv** command which has the following generic form:

```
mv sourceFileName distinationFileName
```

Exercise 1.11. *Juxtapose the the result of the following command with the example of* ***cp*** *above by running both in your terminal:*

```
mv myfile.backup myfile.backup2
```

Hint: Run ***ls*** *before and after each of the above commands.*

1.6.4 Rename?

There is no rename command since the effect of such a command can already be achieved by something we already learned.

Exercise 1.12. *How can you rename myfile to be thefile?*

1.6.5 Deleting a File

To **rem**ove a file we can use **rm** which has the following form:

```
rm fileName fileName2 ...
```

Again be careful with this command since it will not prompt you about if you really want to delete and it will delete it for good. There is an even more powerful and dangerous version of rm when we supply the -r argument[7] which given a directoryname as input will delete the directory and everything in it. Thus (**DO NOT DO THE FOLLOWING COMMANDS**):

```
rm FirstDir
```

and

```
rmdir FirstDir
```

will result in errors but:

```
rm -r FirstDir
```

will silently remove FirstDir/Foo, FirstDir/myfile, and FirstDir/Foo2 forever. This solves the problem we had earlier of how to delete a non-empty directory.

Exercise 1.13.

1. *Change your current working directory to FirstDir/Foo.*
2. *Now use nano to make a file named file1 with a couple lines and use vim to make file2 with a few lines.*
3. *Delete file1 and file2. Can you do it with 1 command?*
4. *Change your directory back to FirstDir and create a new directory in Foo called Bar.*
5. *Add some blank files with the following command:*

```
touch Foo/afile Foo/anotherfile Foo/onemorefile
```

6. *List the contents of Foo.*
7. *Run the following exactly:*

```
rm -r Foo
```

8. *What was the result?*

1.7 Manuals

In this section we will learn how to learn new things about the command line without ever having to leave it! This includes how to learn more about commands we already know and how to find new commands.

[7] r is for **r**ecursive

1.7.1 A help message

Many Unix programs will use one or both of the following to give you a brief help message:

```
programname -h
```

or

```
programname –help
```

This is optional so some programs will have both, one, or neither of the above arguments. Also some programs that have both will have different messages for each. Note that some programs will also use -h to do something other then give a help message. beginex Try them with **nano**:

```
nano -h
```

```
nano –help
```

Now try them with **ls** and **which**.

We have learned that -h and –help sometimes give brief help messages but we also found flaws with consistency. A more consistent system with verbose messages are the programs **man** and **info**.

1.7.2 Man Pages

The **man** program gives you **man**ual pages about what every program you specify. Its general form is:

```
man programname
```

For example:

```
man cat
```

will give you information on the program **cat** which we have seen a few times now. You can even learn more about **man** by running the following:

```
man
```

The interface for reading these pages is the same as **less** so this should feel familiar by now. For those who use C note that you the system calls and libc functions have manual pages in section 2 and 3 of the man pages! There are nine sections to the man pages and sometimes two things with the same name will exist in different sections. For example:

```
man printf
```

will open a manual page for a program called **printf** which is not an important command to learn but merely a good example. We can see that this is from the first section due to the top left corner having this:

```
PRINTF(1)
```

The number in parenthesis is the section the page is in. Thus we get the same page if we instead type:

```
man 1 printf
```

Now those who have ever written a line of C will no that this is the name of a library function to print text to the screen. To view the page for that function enter the following:

```
man 3 printf
```

See how it is a different page?

Exercise 1.14. *View the man page of some of the commands we have learned so far.*

1.7.3 Info Pages

man pages are the standard way of learning how use software on Unix systems and it cannot be stressed enough how important it is to learn how to use. As stated before **man** uses the **less** program to view the pages and thus it has a way of interacting that is similar to **vim**. There is a newer program and set of **info**rmational pages called **info** which instead uses the text editor **emacs** style of interacting. There are some programs that only have both **man** and **info** pages, only one, or neither.
View an info page by running:

```
info programname
```

You will get a full screen viewer like **man** but arrow keys and vim movements wont work! You can move forward pages with “ctrl+v” and backward pages with “alt+v” to view pages similar to **more**. You can move line by line by using “ctrl+n” and “ctrl+p” to move to the **n**ext and **p**revious lines respectively. You can leave by typing “q” just like we did with **more**, **less**, and **man**.

Exercise 1.15. *View the info page of some of the commands we have learned so far. Are there any differences?*

Just like Vim, Emacs is a very powerful editor with a sharp learning curve that is loved by many programmers. We will offer workshops on both Emacs and Vim this year and encourage everyone to learn a bit about both editors.

1.7.4 Programs to help you find commands: apropos and whatis

We can use the **whatis** program to get quick descriptions about other program. The general form is:

```
whatis programname
```

For example we can type:

```
whatis ls
```

and see:

```
ls (1) - list directory contents
```

The 1 should be familiar, it is the section of the man page that **ls** is in. If all of these descriptions are in the computer it would be cool to search for commands like you would use google to search for webpages. That is what **apropos** is for, it has the following general form:

```
apropos whatToSearchFor
```

For example if we wanted to find other editors to use we could try:

```
apropos editor
```

This will produce a list of every command whose name either contains "editor" or whose **whatis** description contains "editor". Clearly this will not always contain the results you want and therefor you can try different searches to refine or expand your results such as:

```
apropos text editor
```

Also note that you can group words with quotation marks:

```
apropos "text editor"
```

Will only return results that have *text* followed by *editor* in the **whatis** description.

Exercise 1.16. *Did you notice that many of the results from searching for editor returned the same descriptions. Try the following:*

```
apropos "Pico clone"
```

and:

```
apropos "Vi IMproved"
```

What did you notice? What do you think is happening? Maybe using ***man*** *will help.*

1.8 Running Your Programs

In this section we will show how to use the command line to run the programs we write. Since no assumptions are made as to how far you are in Western's Computer Science Department's curriculum we will show how to run programs in Python, Java, C++, and C. All code will be provided so you can follow along without needing to know the language. For each one language we will run the iconic Hello World program. Before beginning let's make a workspace:

```
mkdir Programming Programming/C Programming/Python Programming/Java Programming/C++
```

1.8.1 Python

Change your current working directory to where you would like to save your python programs in. For consistency let's use the Programming/Python directory we just made:

```
cd Programming/Python
```

Use a text editor such as Nano or Vim to enter the following in a file called “HelloWorld.py”:

```
print(”Hello World!”)
```

Save the file. Now on the command line enter the following to run your program:

```
python HelloWorld.py
```

This will open the python interpreter with the file “HelloWorld.py” running on it. You should see the following output:

```
Hello World!
```

Exercise 1.17. *One of python’s coolest features is using the shell interactively. Try the following:*

1. *Open the python shell with the following command:*

   ```
   python
   ```

2. *Type the following and hit enter:*

   ```
   print(”Hello World!”)
   ```

3. *What happened? How is this different then what we did before with “HelloWorld.py”?*
4. *To leave enter:*

   ```
   exit()
   ```

Using the shell this way can be great for learning new features and debugging code.

1.8.2 Java

Change your current working directory to where you would like to save your java programs at. Again we will use the directory we just made so assuming we just finished the python step we will enter:

```
cd ../Java
```

Create a file called “HelloWorld.java” with the following content:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Now compile the program with the following:

```
javac HelloWorld.java
```

If errors occurred during compilation they would be displayed during the last step. You would need to fix the lines in the file causing the errors and rerun the previous line until things compiled correctly. Note, assuming you copied everything correctly, you will see the file "HelloWorld.class" when you enter:

```
ls
```

Now we can run the program with the following command:

```
java HelloWorld
```

Which, as you might expect, will display the following:

```
Hello World!
```

Exercise 1.18. *To see example of how errors will be displayed edit "HelloWorld.java" to no longer have class on the first line. Recompile with the* ***javac*** *command we just learned. See how the top line that is returned informs us that java was expecting the world class but couldn't find it?*

1.8.3 C++

As before change to our C++ directory with:

```
cd ../C++
```

With an editor make a file called HelloWorld.cpp with the following content:

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

int main() {
    cout << "Hello World!" << endl;
}
```

C++ is a compiled language which means that we need to first compile it before we can run it. We can compile our new program with the following command:

```
g++ HelloWorld.cpp
```

Any errors will be displayed but if you copy my lines exactly you will see nothing except for a new prompt on the screen. List the contents of your current working directory to see a new file called "a.out", this is the file which is our new program. We can run it with the following:

```
./a.out
```

The output should be exactly what you expect.

Exercise 1.19. *C++ defaults to compiling our program into a file called a.out but we can specify what we would like our program called with the following option to g++:*

```
g++ ourprogramfile.cpp -o whatIWantMyProgramToBeCalled
```

Use this to compile "HelloWorld.c" into a file called "HelloWorld". How can we run this new program?

1.8.4 C

As before change to our C directory with:

```
cd ../C
```

With an editor make a file called “HelloWorld.c” with the following contents:

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

We can compile our C program with the compiler gcc using the following command:

```
gcc HelloWorld.c
```

This has very similar semantics to g++ including the same way it compiles to a file called a.out and how you can change this in the same way we learned in the C++ exercise. This is because they are essentially the same program. We again can run our new C program with the following:

```
./a.out
```

And that is the basics of how to run programs using the command line. There are many extra options to these commands but this minimum will allow you to make most programs.

1.9 Shortcuts

There are a number of shortcuts that save typing time and allow you to remember less. We will discuss a few now.

1.9.1 Reuse previous commands

You will notice the more you use the command line that there the same sequence of commands are used a lot. For example when you are writing a program you will edit it using nano, vim, or emacs and then compile or run the program. You will then notice a bug and open the editor again and then test your changes by compiling or running your program again. Such as:

```
vim HelloWorld.c
```

```
gcc HelloWorld.c
```

```
./a.out
```

```
vim HelloWorld.c
```

And this will cycle will keep going. Wouldn't it be nice to not have to keep retyping these same commands? There are a few tools for this and we will discuss two of them.

The simplest is to use the up and down keys to look through your history of commands. By pressing up once you will see the last command you entered. You can continue hitting up to see later and later commands. In our example instead of retyping "vim HelloWorld.c" we could have hit up three times at the prompt.

The only issue with this method is sometimes the command you want is far back in your history and the amount of time to press up till you find it is longer than if you would have just typed it again! This is a common mistake even by upper class men. A solution to this is the following:

```
!!
```

will also run the last command and is equivalent to pressing up once. More importantly though is this feature:

```
!str
```

Will find the last command you entered that began with str. In our previous example we could have entered:

```
!v
```

which will run:

```
vim HelloWorld.c
```

since it was the last command we entered that started with a "v". Sometimes you will need to use more than one character such as this situation:

```
vim HelloWorld.c
```

```
view HelloWorld.c
```

```
!vim
```

Not that since we wanted to run vim again we had provide the three characters we did. If we had used “v” or even “vi” it would have matched to the view line instead.

1.9.2 reverse search

Another way to go back to previously entered commands is by pressing "**Ctrl + r**" while at the command line prompt. This will search for any past commands you've entered matching the string that you type in.
As you type, you will see the first command that matches appear. If that is not the command you are looking for, you can either continue typing in hopes that another previous command will appear or press "**Ctrl + r**" again to move to the next matching command in history.

Exercise 1.20. *Try it out: Press "Ctrl+r" and then start typing "cd" to see the last change directory command you entered. To search through all of your past change directory commands, continue pressing "Ctrl+r".*

1.10 Remote access through SSH

You can access another Unix machine from the command line with the program **ssh**. Suppose you want to work on the files for a project that you started in Westerns CS lab computer from home using your Mac. You can run the following:

```
ssh -p 922 username@linux.cs.wwu.edu
```

Where you replace “username” with your username. The first time you use it from a computer you will need to type “yes”. Then you will need to enter your password for the machine and useraccount your trying to log into. You will then be logged into a linux computer from the school.

Note that this implies that although you may think you are the only one using a computer at school there could be many other students who are also using it through ssh.

There is also programs such as *putty* which allows you to use a Windows computer to access a Unix machine. To learn more about accessing the lab machines at Western follow this url:`https://support.cs.wwu.edu/index.php/Accessing_Linux_with_SSH`

Note the section on **scp** which allows you to use ssh to grab files from a remote computer. This can be useful if you need to grab a file from the linux machines on campus but you would like to use your own computer to work with it.

1.11 Retrospective and Closing Comments

If you understand all the content that we just covered then you can now do all of your programming and file managing from the command line. Try to use the command line on your next few projects to master these skills. We will offer the sequel to this workshop later in the quarter that will expose you to some very powerful commands and concepts and allow you to do tasks that would be very hard if not impossible using a graphical user interface.

If you are curious and eager to learn more about the command line then we encourage you to use the **man** pages and your favorite search engine to look into the tools and topics that we just outlined. Also note that you will learn a lot from friends, teachers, and colleagues in the years to come if you continue in the CSCI program.

Please take our survey once the presentation has ended to help us improve the Peer Lecture Series: https://www.surveymonkey.com/r/CHJC7ZW

Bibliography

[1] Figure 1 found at http://www.linuxtrainingacademy.com/wp-content/uploads/2014/03/linux-directory-tree.jpg