# *COMP 161*
# *Lecture Notes 03*
# *Working with the Shell*
*January 8, 2018*

In these notes we'll dig deeper into the CLI and see how working with *bash* gets us ready for the bigger changes in mindset we need to work with C++.

## *Essentials For This Class*

There are a few things not discussed in your CLI tutorial that merit some discussion as they are integral to doing anything in this class.

### *Submitting assignments with handin*

A script written for us by a former CS professor[1] here at Monmouth allows you to submit most of your assignments via the server. The name of the script[2] is *handin*. Like many CLI commands, its usage is documented in help text accessible with the *-h* option like so[3]:

[1] Thanks, Don Blaheta!

[2] and the command

[3] Go read the documentation. Now.

```
handin -h
```

Submitting files is done by listing the course and assignment name as arguments along with the names[4] to the files you're submitting.

[4] path

```
handin course assignment file[s]
```

We'll almost always just submit a directory containing all the files we wish to submit. The handin script will zip these up for easy processing. In the first lab you'll run this command to submit the folder lab1 as assignment lab1 for the course comp161[5].

[5] the second lab1 is the directory name

```
handin comp161 lab1 lab1
```

It is possible to retract a submission from handin as long as the instruction hasn't collected that assignment. You should make certain that only one set of files is submitted per assignment. I usually get the most recent submission, but sometimes not. If not, and your grade is lower as a result, it might not get corrected as its your responsibility to hand in the correct work. So, I highly recommend you check the handin help text for how to work with previously submitted work.

### *Emacs Commands*

Emacs is the text editor we'll be learning in this class. Commands usually require you to combine some keys with the *ctrl*[6] key or the meta key[7]. For example, the command to close Emacs is written *C-x C-c*. That means, "press and hold ctrl then x, then release them, then press and hold ctrl then c, and release them." It should feel like your rolling through keys starting with ctrl. If you're familiar with the windows command *ctrl-alt-del*, then you know what I'm talking about.

[6] shown as **C** on the sheet

[7] Shown as M. See below.

If you're at the CLI, you need two things really:

- *To launch emacs*: emacs
- *To open/create a file with emacs:* emacs *filename*

Once you're in Emacs you'll need at least these three emacs commands.

- *To start the tutorial*: C-h t
- *Save current file:* C-x C-s
- *Close Emacs:* C-x C-c

The tutorial will walk you through a host of other essential Emacs commands and is part of your first lab and homework assignment. Like the CLI, you can fight the Emacs way of doing things or you can buy-in and take the time to learn and use the commands. When you do buy-in, you'll find that Emacs is insanely powerful and will save you a lot of time and frustration down the line. Professionals use it for a reason.

### *Emacs Meta Key*

If you're on a linux or windows machine, then you have an *alt* key. That's your meta key. So commands like *M-b* are telling you to press and hold alt then b, then release both. If, however, you're on a Mac, you lack and alt key. You have two options[8]: use the *Esc* key or tell your terminal to use *option* as the meta key. If you go the route of *esc*, then I don't believe you hold the the key down[9].

[8] `http://stackoverflow.com/a/3566557/1042494`

[9] I could be wrong about that.

## *Paths*

Much of what you do in the shell will deal with specifying the location of a file or folder on the server via it's **path**. In fact any time you reference a file or folder in a command you can use a path instead. This means you can pretty much work on any file in the computer

from anywhere else on the computer. Realizing this can make you life much easier. For example, if you need to copy a file from folder *A* to folder *B* but you're in folder *C*, then there is zero reason for you to switch either folder *A* nor *B*. This is hard to grasp coming from a GUI environment. You can't click on a file if you don't have the file explore open to the directory containing that file. This spacial limitation does not exist on the CLI.

Paths come in two flavors: *relative* and *absolute*. Understanding the differences between these path variants and how to use, or spot, one versus the other is an important part of life at the CLI.

### *Absolute Paths*

Absolute paths always begin from the main directory of the file system, *root* or /. For that reason they're easy to spot:

> When a path begins with **/**, then it's an absolute path.

For example, everyone has a home directory on the system and all the home directories are found within the *home* directory. The home directory is in turn, housed within /. So the absolute path to the home directory for user *jdoe* is:

the ending / on the path is optional. I like it because it makes the fact that we're explicitly dealing with a directory

```
/home/jdoe/
```

Absolute paths are great because they unambiguously specify a file or folder on the system. Dr. James Logan Mayfield, IV is my full, absolute, name. You're unlikely to get me confused with anyone else if you use it. On the other hand, that name, like absolute paths are often long and unwieldy. This makes them hard to type[10] and often requires some big picture understanding of the file system's overall organizational structure.

[10] TAB autocomplete solves this problem!

### *Relative Paths*

Relative paths specify a path relative to your current working directory[11]. Put another way, the absolute path to your working directory is an assumed prefix to the absolute path of the file or folder in question. There are a few ways to recognize and write relative paths. The first is, in my feeling, more explicit and therefore less prone to ambiguity. The path to your current working directory can always be invoked with the shortcut **./**[12]. This leads to your first indicator of a relative path:

[11] the *pwd* command tells you what that is

[12] read ./ as "here"

> When a path begins with **./**, then it's a relative path.

So if my current directory is */home/* then we can form the relative path to jdoe's home directory like this:

```
./jdoe/
```

It turns out that the ./ is optional and this leads to the other typical way of picking out a relative path.

> When something shows up where a file/folder path specification is supposed to be and there's no leading /, then it's a relative path.

If we are once again working out of */home/* then *jdoe/* is a valid relative path to jdoe's home directory. In general, I like using ./ because it's clear you're providing a path. Leaving off ./ leaves it to the reader to decide the thing they're about to read is a path[13].

[13] thankfully when the reader is the OS, you tend not to have problems

### *Path Shortcuts*

In addition to the ./ shortcut to the current working directory, there are a two path shortcuts you should memorize.

1. **../**

   This shortcut always refers to the parent of ./. If you're in your personal home directory, then ../ is */home/*. If you're in */home/*, then ../ is /. The odd directory out is /. It has no parent, so the system treats it as its own parent. That means that relative to root, ../ is still root. So, when you're not in /, *cd* ../ is like hitting the up button[14] in your GUI.

[14] not necessarily the back button

2. **~**

   This shortcut is your home directory. So *cd* ~ is the command to "go home".

It's worth noting that adding *-a* to *ls* adds **.**[15] and **..**[16] to the list of directory contents.[17]

[15] here
[16] parent of here
[17] try it out!

### *The Power of Paths*

As stated earlier, it is possible to run commands on files that are not in your current working directory by using their path in the command. In fact, anytime you specify a file or folder name without its path, you're actually give a relative path to a file in the current working directory! Compare this to working in the GUI. You typically have to navigate to the folder containing the file you wish to work with, and then select that file. In a CLI environment you can run a command on any file on the system from any directory on the system[18]. This kind of capability is necessary for copying files. If you want to copy file *a* to your current working directory and *a* is in your current directory's parent directory then you can run

[18] assuming you have the proper file permissions

```
cp ../a ./
```

. What you might not realize is that you can also do things like this

```
cp ../a ./sub/folder/over/here/
```

. Notice that neither the target file nor the destination is the current working directory!

If you're going to repeatedly work on a one or more files in a specific directory, then it makes sense to be in that directory. If, however, you're running a one off command involving files not in your working directory, then resist the urge to first change your working directory before running the command. When you tap into the power of paths, you can save a lot of time at the CLI.

## *Racket Functions and CLI commands*

If you made use of the interaction window in Dr. Racket, then you're setup pretty well for the basics of the CLI. Let's review what do we know about using Racket functions at the interaction window:

- They use *prefix* notation in which the operator[19] comes before the operands[20] and everything is separated by white space.
- Function invocations are surrounded by parenthesis.
- Racket functions have one or more parameters and the number of parameters for a given function is fixed. Additionally, the order in which you pass parameters matters.
- Racket functions take data values as input and return them as output, always. Given the same input, a Racket function will always produce the same output.[21]

[19] command name, function name, etc

[20] arguments/inputs

[21] This is the essential property of functions

The question we now ask is, in what ways are CLI commands similar to and different from Racket functions?

Here's what we'll learn:

- Bash commands also use prefix notation and white space to separate the command name and its arguments
- Bash commands are not surrounded by parenthesis
- Bash commands can have zero or more parameters and many commands have optional parameters. This means that one command can take a variable number of parameters
- Bash commands don't always produce output at the CLI. Sometimes they produce a *side-effect* on the system that we can't see unless we look for it. Some commands will produce different results on the same input, or they have different behaviors based on the *state* of the system.

In the end they're both REPL interfaces. The big differences all stem from the fact that most of the commands we use on the CLI are not *functions* but imperative procedures that interact with the ***state*** of the computer in some way. We'll explore this idea in detail with some examples. It's a very important concept and one that will form the basis for a lot of our work in C++. Pay careful attention as you read ahead.

### *The Environment and State*

Run the command *env* at the CLI. What you see is a list of STATE VARIABLES and their values. These variables are specific to you and your session. For now we want to look at three of them: *HOME*, *PWD*, *PATH*. The first stores the path of your home directory, the second your current working directory, and the last lists all the directories where the computer should find the commands that you type. The values of these variables determine a part of the *state* of the system and thereby effect the results of certain commands. To see their current value run the command *echo $VARNAME*. For example, *echo $PWD* will print out the value of the PWD variable[22].

Racket functions[23] are stateless. Their action is strictly determined by their inputs. Given a particular input they will always produce the same output. They are completely predictable in this manner. Statefull procedures like most of our CLI commands have results that are dependent on their inputs *and the state of the system*[24]. This means that a command can produce different outcomes with the same argument if the state of the system has changed.

[22] Go print out all their values now

[23] at least the ones you wrote in COMP160

[24] the current values of these variables and things like the contents of the hard drive

### *Function Inputs and Outputs vs. I/O*

In a functional world we often talked about the inputs and outputs of a function. More formally, we might call these the function arguments and return value. This is very different that the kind of INPUT and OUTPUT we'll deal with on the CLI and in C++. When Dr. Racket printed the return value of a function it was implicitly carrying out an output operation that displayed the return value to the interaction window. Similarly, when Dr.Racket read the code typed at the interaction prompt, it was carrying out an implicit input operation. You, the programmer, didn't need to explicitly cause the program to read code from the prompt and print results to the prompt. Put another way, you directly requested something happen on the monitor or keyboard. This was all do for you by the REPL code.

A REPL encapsulates a very basic computational interface: read *input* from the user, compute something based on that input, then

*output* the result. You've been interacting with I/O[25] implicitly. Now you'll do so explicitly. This can be a tricky thing to get used to. The "input" and "output" from your functions are not at all the input and output we'll be dealing with at the CLI and in C++.

[25] Input/Output

### *Bash examples*

Let's see how interacting with STATE, INPUT, and OUTPUT all plays out with some of the most common bash commands.

1. *cd directory*

   What happens when you invoke the change directory command? First the system goes and looks in every direction listed in your PATH variable to see if there is a command named *cd*[26]. Once it finds it, it executes that command with the arguments. If the argument is a relative path then it is effectively appended to the the value of your PWD variable to get an absolute path. Already, we see that even the basic execution of the command is dependent on system state.

   [26] type *which cd* to see the absolute path of the command

   Now what's the result of cd? We get no output. This is because the *cd* command is a MUTATOR. It's purpose is to change the value of a state variable, namely PWD. Mutators are an important class of procedures for the style of programs we'll be writing in C++. Once you have state, you tend to want to change it and mutators provide a consistent means of abstracting this action.

2. *ls*, *ls -l*, and *ls -la*

   Here we see three variations of *ls* with zero, one, and two arguments respectively. What you probably don't know is that *ls -la* and *ls -al* are both allowable and equivalent. So not only can we have a variable number of arguments, but order doesn't necessarily matter! You probably didn't have this kind of flexibility in your Racket functions. You could write functions that would allow this, but it takes a lot of extra work.

   The execution of *ls* interacts with PATH in the same way that *cd* does[27]. The *ls* command is clearly not a MUTATOR. No variables change. It does, however produce some output and that output is not solely dependent on the arguments. The arguments control formatting, but at the end of the day the output is really driven by the contents of your current directory. This makes *ls* an ACCESSOR procedure. It retrieves the value of the PWD variable, and then displays the contents of that directory[28].

   [27] all commands do, so we'll stop talking about it

   [28] so it's also dependent on the state of the file system itself. Not just PWD.

3. *echo $(ls)*

The output of *ls* is not functional output. It's just like the output produced by Dr.Racket's REPL. The command produced a string value and then the CLI printed it to what's called the *standard output*[29]. The above command is an explicit OUTPUT command for printing the result of *ls*. What you'll see is the raw unformatted string value returned by the *ls* command. Notice you lose all the nice formatting that the CLI injects[30] when it prints. If you wanted it back, you'd have to put it in yourself and in doing so rewrite part or all of the *ls* command. How does this work? The *echo* allows the programmer to explicitly produce printed output. By surrounding *ls* with *$()* we invoke command substitution which takes the result of the command and places it inline on the CLI. This is the equivalent of nesting functions in Racket[31].

[29] the screen/CLI prompt

[30] because of some state!

[31] (echo (ls))

With just a few commands we've encountered several classes of procedures that we'll be designing and implementing in C++: MUTATORS and ACCESSORS facilitate standard interactions with STATE VARIABLES. OUTPUT procedures write values to the *standard output* giving the programmer control over what gets printed, or written, to the screen. Mutators and output procedures are particularly notable because their purpose is to produce a SIDE-EFFECT. A mutator modifies state, which in turn can change how other commands behave. It affects other parts of the computer. An output procedure writes information to an output device where there is none. Strictly speaking, it has not functional value[32]. To really get a feel for I/O based programming we should really look at redirect though.

[32] in the Racket Function sense

### *Redirects*

Your tutorial introduced several redirects. These all modify the expected I/O behavior of CLI commands.

1. | redirects output written to the standard output to the the standard input, where it can be read by he command on the left of the redirect. What would have been written output is now something that appears to be text typed by the user for the command on the left hand side of the redirect.
2. > redirects text that was to be written to the standard output to a file instead.
3. >> like > but with a variation on the write effect (append vs overwrite)

The < redirect is a bit different as you're not really redirecting an effect as much as you're causing one. That is, the name of a file is not an implicit command to write to stdout, and so using < is probably

best though of as a compound effect: read from a file and write its contents to the standard input.

This is really just a description of the < redirection uses we've seen. Others exist.

By using redirects we can start to build COMPOUND OPERATIONS. Just like we can nest function calls in Racket so that the output of one function is the input to another, we can redirect the output of one Bash command to another. Hello programming.

### *Expansions*

Expansions tap into the core ideas of functional input and output in that they allow you to substitute one value for another within a particular command just like we can substitute the return value of a function for it's function call when evaluating a Racket expression. The key here is that we're thinking about VALUE SUBSTITUTION as opposed to some kind of effect redirection.

Shortcuts represent a basic form of expansion. In some cases they're as simple as named values. For example, for the user jdoe, the expression  has the value of */home/jdoe/* and we can pretty much use the former anywhere we want the later. The short cut for here, *./*, and parent, *../*, are a bit more complex as their value is dependent on the PWD state variable.

One of the most useful expansions is wildcard, * expansion. Here the expressions expands to all values which match a specific pattern. The pattern *./*.pdf* expands to the path to all of the pdfs in the current directory. If you have 32 pdfs in your current directory, then you get 32 paths! This pays off huge for things like copying. The command

```
cp *.pdf foo/
```

will copy every pdf in the current directory to the sub-directory foo. Wildcards are so powerful that you should be careful using them at first. They can cause commands to do way more than you expected if you and the computer don't interpret the pattern in the same way. Using wildcards along with *rm* is a really good way to accidentally erase some files.

Brace expansion, like wildcard expansion, expands to everything that matches the pattern. The pattern {*lab1,lab2*}*.cpp* will expand to *lab1.cpp lab2.cpp*. Combine this with wildcards and you can create some pretty powerful patterns. For example, *.{*cpp,h*} expands to all the cpp and h files in the current directory. This kind of pattern is likely to pop up a lot when we start C++ programming as these two file types are used in C++ programs.

Parameter expansion and command expansion are, given our background, significant CLI tools. *They let us recapture the functional*

*input and output we know from Racket.* In Racket we'd write things like $(f(g5))$ and expect the VALUE output by $(g5)$ to be fed for $f$ as an input. To get the same thing in bash, we use command expansion:

```
f $(g 5)
```

Now stop and think, How is that different than this command?

```
g 5 | f
```

The bash command $f\$(g5)$ is not a redirection. It takes the value of the output of $g5$ and uses it as the input to $f$. On the other hand, $g5|f$ takes what $g5$ writes to STDOUT and instead causes $f$ to read it from STDIN. The result in this case might be the exact same thing, but the first route seemingly avoids notions of I/O and instead uses *functional computing*.

Basic parameter expansion let's us easily substitute the value of a variable for its name. Put another way, it's an accessor shortcut. Try this:

```
echo PWD
```

What you should see is *PWD*. You might have expected to see the same thing as the command *pwd*, why? In Racket, feeding a variable to a function meant "use the value associated with name". The PWD variable is a different beast and so we have to be more specific. Try this:

```
echo $PWD
```

Now, we see the same thing as *pwd* because the parameter expansion invoked by **$** effectively retrieves the value associated with the variable *PWD*.

All of these expansions let you recapture some of the functional feel of programming in Racket. The alternative is to chain together effects through redirects.[33]. More practically, they're the gateway to some serious commandline-fu.

[33] This is subtle and very very important. Give it serious thought

## *Big Picture*

These notes merit careful study. The interaction of commands with state variables and I/O illustrate fundamental principles in computing, principles that we'll study and utilize when we program in C++. One perspective of the procedural, imperative style that we'll be using in C++ is that it's largely about interactions with state. You'll quickly see this play out on both the micro and macro level.