Lecture 03: Bash Basics

Bash

The last lecture mostly covered Unix and 'Unix Commands'. It is a bit difficult to disentangle the difference between a 'Unix Command' and a 'Bash Command' because the two are sort of the same. The main difference is that when you are talking about Bash you are usually talking about Bash-scripts which are files which contain a series of Unix commands…

Actually, that isn't precisely true. The original Unix shell 'sh' was like a very stripped-down version of Bash. Many of the conveniences of Bash simply were not there and as you can imagine, people kept adding things. So, there are definitely things which are 'Bash' but not 'Unix'. However, this means that Bash has 'all the feels' of an evolved program (rather than a designed one). Think of a platypus with its eggs, poison, electroreception, and phosphorescence. Now imagine someone stitched a platypus together with half a dozen other animals, plants, robots and… you get the idea. What this means is that Bash will often have multiple 'solutions' to the same problem. Moreover, the syntax is not clean like other languages. Bash can be very difficult to read (especially the way it uses brackets – we will see later).

My First Bash Script

Usually programmers like to start with 'hello-world' like programs and this will be no exception. Except, we are starting from a terminal. You probably already know how to use 'mkdir' (at least I hope so), so you can pick a sensible place to start.

Command one. Create the script.

```
touch my_script.sh
```

'touch' is a convenient script. 'touch' is not strictly for making files, but in true Bash fashion it doubles as a file-maker. Strictly speaking 'touch' is designed to prod a file to update when it was last accessed (but not modified). However, if the file does not exist, it creates it… (to this day I still find this a bit baffling).

Command two. CHMOD!

```
chmod +x my_script.sh
```

So normally when you create a file it is not executable. You have probably not run into this issue when programming in C++ as you write a .c file and then compile a '.o' or '.exe' file. The compiler (g++) is smart enough to know that the output should be executable and sets the access flags thusly. However, by default the .c file is merely read/write but no execute. So, if you create a Bash script, because it is the file being executed you must **chmod** it.

Command three. The first line.

The first line of any Bash script should be:

```
#!/bin/bash
```

Surprisingly, you will mostly get away with omitting the first line. That is because the default shell on most Unix systems is either already Bash or something akin to it. This first line can be replaced with other things like zsh or even python (if you are running python, here it will not work for your purposes). Note there are no spaces. The 'she-bang' (#!) is a recognised command. '#' by itself is

usually a comment character in Bash. The '/bin/bash' part is literally the location of the Bash executable on most Unix systems. You can 'cd /bin' to go looking for it if you want (I suggest you do).

Long story short, this line tells the shell, this is indeed a Bash program and to use Bash as the interpreter.

The second line is the hello world line.

```
echo "Howdy World"
```

'echo' is basically the print-to-screen command of Bash. It has some nuanced rules about it (especially quotation marks – to be covered later) but it generally can be thought of as 'cout' (from C++) or print (from other languages).

The last part is to execute the script, something you should be familiar with.

```
./my_script.sh
```

The './' bit might make a bit more sense now. This is telling the shell to run the 'my_script.sh' which is in the current directory ('.'). Otherwise, it would go on a merry quest to find a my_script.sh pretty much anywhere on the path (if you don't know what this means it will be covered later).

My Second Bash Script

So… this is a bit silly, but I think it is a good place to start in Bash. Bash programs are often used to run other programs rather than really doing anything themselves. The idea of the next script is to write a script which creates a file (a .sh file), puts some text in that file (coincidentally some text that will basically make it a hello world program), then execute that file…

```
#!/bin/bash
touch hello_world2.sh
chmod +x hello_world2.sh
echo '#!/bin/bash' >> hello_world2.sh
echo "echo Hello World 2" >> hello_world2.sh
./hello_world2
```

As I said, a bit silly but it has a few important parts. Number one, it is a reminder of the need to adjust the file permissions (this will be important for Assignments). Number two, it shows some redirection. Instead of printing to the screen 'echo' is now appending to the specified file. Number three, it is a Bash script running another Bash script. This is important when thinking in Bash. Programs in Bash are a bit more akin to functions than they are to programs in high level languages like C++. Now, before you get very confused, yes Bash has its own idea of functions (more on this later) but the point is that when you program in Bash, you are probably gluing a lot of other programs together and some of them may be programs you wrote yourself.

Note: I think of Bash like glue. It is for connecting other solid things together. I would never make a house out of glue…

Arguments

The function analogy is particularly helpful when you realise that every Bash function (well 99% of them) take arguments, something high-level programs sometimes do, but not always. However, unlike C++ Bash doesn't really care about arguments in the same way, treating them rather… optionally?

To test this, you can arbitrarily add arguments to your 'my_script.sh' or 'hello_world2.sh' script.

```
./my_script.sh a series of arbitrary arguments duck 5 duck
```

This will work. It will not do anything, but it will not crash either. We can make better use of this functionality by gaining access to the arguments in question. Try the following code:

```
#!/bin/bash
echo $#                 # Number of arguments
echo $1                 # First argument
echo $@                 # Whole command line
```

There are three in-built ways (I'm sure there are more) of talking about arguments. Each requires the use of the '$' symbol. $# returns the number of arguments (very useful), $1 returns the first argument (#2 the second and so on… $0 returns the name of the script… but why?). $@ returns all of them… which is a bit odd but will be useful later when we discuss **for loops**.

Another useful tool is the keyword 'shift' (you might want to read about loops first).

Have a look at output of the following code when given multiple arguments.

```
#!/bin/bash
echo $@                 # Print arguments
shift
echo $@                 # Print again (where did $1 go?)
```

It will remove the first argument entirely and shift the second (and third etc.) arguments up one spot. This is very convenient for parsing arguments as you can deal with one at a time. What makes it even better is that you can deal with arguments 'X' at a time. For example, if you get a "-n" flag perhaps you need the next two number "-n 21 51" but if you get a "-q" flag you only need the "-q". With shift you can get rid of 3 arguments in the first case (shift three times) and 1 argument in the second (shift once).

Parsing arguments is an important part of writing Bash scripts. It can also be frustrating as there is a lot of work to make a script that still doesn't do anything useful other than understand it has 'options'.

If you start assignment #1 not knowing about **shift** it is going to be much harder than it needs to be.

Debugging

Before we continue, I would like to mention debugging. Debugging in Bash… (cries).

Unlike other languages where syntactic mistakes are usually picked up by the compiler and prevent further progress, Bash tends to steamroll through things that do not work and completely ignore them. While the C++ compiler can be frustrating, it is usually helpful at identifying when something went wrong and roughly where it happened. Bash can often just… ignore problems entirely.

Sometimes you will get an error message for particularly egregious errors, but sometimes you will get literally nothing. Bash turns syntax errors into bugs!

One way to get around this (a bit) is to run Bash in explicit mode. This can be done by invoking the Bash executable.

```
/bin/bash -x my_script.sh
```

Firstly, this command is running bash just like you would run… cd or ls. Normally bash takes one parameter (the script name). Here it has an extra flag. The '-x' flag will cause the script to print out every line it receives and then what it produces as a consequence. It isn't the best debugger in the world, but it is sometimes better than nothing. As an exercise I'd recommend running it on 'My Second Bash Script' (whatever you called it).

Basic Programming Logic in Bash

I do not really consider any program without an if statement to really be a program. If a program is not making decisions, it is not really doing anything. So that is where we will start.

If Statements

```
#!/bin/bash
if [ $1 -eq 1 ]
then
      echo "True"
else
      echo "False"
fi
```

This is the 'clunky' square bracket version. This program basically asks… is the first argument ($1) equal to 1. The reason I use the word 'clunky' is the abhorrent reliance on white-space (space, tab etc) that Bash has. The spaces before the '$1' and after the '1' are syntax (leave them out at your peril). You will soon learn (to your misery) that Bash is very sensitive to **white-space** (also it over-uses special characters). White-space at the start of a line is ignored, so at least you can do *some* formatting for readability. After the first line, the rest is fairly self-explanatory.

If you are a 'same-line-**if**er' you can use the following instead:

```
if [ $1 -eq 1 ]; then
```

Now, it is important to understand that what goes between these brackets are not exactly trues and falses. For example you can write:

```
if [ ls ]; then
```

What does this do? Well… it runs 'ls', then sees if it crashed or not before returning true if it worked and false if it did not. Given that I've never had 'ls' fail, this always returns true. I originally found this a bit strange because you can write all manner of nonsense in the brackets and just not get what you are expecting. For example:

```
if [ 0 ]; then
```

I'd normally think this was false… (it is in C), but it doesn't work that way. Again, the line '0' is a line which executes without failing. It does not do anything, but it does not crash. When we start writing C programs, we will be able to take advantage of this functionality.

Of course, this is all very clunky, especially when it comes to AND and OR conditions. For example, a fairly normal thing to do would be to say if a value was between two other values do something.

```
IF a > 5 AND a <= 10  # This is not bash code
```

In Bash (using this syntax, there are other options) we can do something similar.

```
if [ $1 -gt 5 ] && [ $1 -le 10 ]; then
```

Now for the next part, I am going to show you something quite… awful and then show you something better. So do not panic. The point is just to be mean to Bash. If you want to do something a bit involved.

```
IF ( a == 1 AND b > 5 ) OR ( a == 2 AND b > 10 ) # Not Bash code
```

This is not the nicest code, but I have written things that are akin to this quite commonly. The Bash equivalent can be (this is not the only option).

```
if [ \( $1 -eq 1 -a $2 -gt 5 \) -o \( $1 -eq 2 -a $2 -gt 10 \) ]
```

The '\' characters are necessary as escape characters (like '\n' being new line) to ensure the interpreter understands that it is a bracket for the purpose of precedence. As you will probably agree… this is… just the worst. Fortunately, there are alternatives.

You could separate the conditions better.

```
if [ $1 -eq 1 -a $2 -gt 5 ] || [ $1 -eq 2 -a $2 -gt 10 ]
```

This is better (and more familiar) but it still isn't super readable. Moreover there are limits to this kind of syntax. Below is something more flexible.

```
if [[ ( $1 == 1 && $2 >= 5 ) || ( $1 == 2 && $2 >= 10 ) ]]
```

Yes… double square brackets are a thing. It is like single square brackets but 'smarter'. Not only is this code cleaner, it is also more reminiscent of C++, Java and other languages. Unfortunately, this is not where it ends. You'd think by now 'ifs' would be over. Yes, I have neglected to mention 'elif' (which is just else if and it also requires a then), but that is not what I'm talking about.

We can also do a 'mathy if'.

```
if (( 1 == 1 ))
then
      echo "One Equals One"
else
      echo "Math has ceased to make sense to me"
fi
```

This almost looks like something from another language (minus the superfluous double round brackets). This is probably a good place to sit down and cry. Not because your code will not be unreadable to you, but rather because different people choose different styles and unlike other languages Bash provides far too many options which are very different from each other. Hence, Bash is often difficult to read. Unfortunately, 'for loops' aren't much better.

In quick summary, there are single square brackets, double square brackets (which are like a better version of single square brackets) and double round brackets. There are three syntaxes for the same thing (there may be more).

For Loops

The first **for loop** syntax I found when researching Bash was the following:

```
for p1 in `seq 1 10`
do
      echo $p1
done
```

This makes sense but does not seem very flexible. 'p1' works similarly to the input arguments in that we need to use the '$' to access it (for reference all variables work this way in Bash). Moreover, the apostrophes are very strange ones (you usually find them on the ~ tilde key, isn't it nice that we have so many options for apostrophes…).

```
for (( i = 0; i < 20; i++))
do
      #Code
      echo $i
done
```

This is far closer to something from another language. Again, we are using double round brackets. In this case, we could use $i to get the value of 'i'. This also allows a lot more flexibility in the condition as we can get away with a lot more math-like expressions within curly braces.

There are also array versions. For example, you can define an array and then iterate over it.

```
choices=( R P S L V )
for p1 in "${choices[@]}"
do
      echo $p1
done
```

Perhaps I've jumped the gun doing array definitions before variable definitions. How to define a variable:

```
a=1
echo $a
```

With that out of the way (note the lack of **white-space**… "a = 1" will not work… only "a=1"), we should be able to parse the for loop above. We define an array of five characters. We then loop over that array setting p1 to the value in the array as we go. We'd expect it to print (via echo) R, P, S, L, V on separate lines. The syntax leaves a lot to be desired though. Quotation marks (type them manually rather than copy-paste as they may be the wrong ones) with a curly brace inside and the square brace with an '@' symbol. That is some pretty dense syntax. Fortunately, it is pretty good syntax for looping through input arguments:

```
for p1 in "#@"
```

```
do
      echo $p1
done
```

The curly braces can also be used for simple lists

```
for p1 in {1..10}
do
      echo $p1
done
```

There are also square-brace equivalents for while.

```
COUNT=0
while [ $COUNT -lt 10 ]
do
      echo $COUNT
      let COUNT=COUNT+1
done
```

And even a until (the opposite of while).

```
COUNT=20
until [ $COUNT -lt 10 ]
do
      echo $COUNT
      let COUNT=COUNT-1
done
```

Finally, Bash does support 'break' and 'continue'. For those unfamiliar, 'break' just stops the loop in its tracks (skip to the end). This is useful when looking for a suitable candidate from a list (like in many searching algorithms). Continue sends the loop to the next iteration (skipping any remaining code within the loop). This is a slightly neater way of just skipping a single iteration (perhaps if a member of a list is invalid for some reason).

Final Remarks

I would recommend picking a style and staying with it as much as possible. You might be tempted to go with double-round braces ((())), but remember that square braces also have a few interesting capabilities. For example you can use [-d my_directory] to test if 'my_directory' exists and is a directory amongst many other options. The point is to be familiar with them and play around with them until you feel comfortable about how they work (remember **white space is syntax**). Another minor hint is when copying from online/lecture slides, sometimes you will get the wrong dash character and that is another headache. Yes, '-' is different from '–'.

You should be able to write an isLeapYear program (I suggest using double round brackets for the math). For a bigger challenge you should be able to write the same program that takes an arbitrary number of years (i.e., 1955, 2028, 5400 etc) and then prints out the number of leap years or something similar.

To test your for-loop knowledge I would recommend writing the chess board program. Basically, it takes a number and then prints out something like this (for 3). You should ‘man echo’ to find out how to echo something without a new line.

```
B W B
W B W
B W B
```