Lecture 04: Bash Less Basics

Variables

We covered this very briefly last lecture, but variable assignment is usually a fairly trivial exercise. So, you might wonder why I did not cover it first. Well… in Bash it is a bit stranger than you would expect. Of course, we have the trivial example:

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

This is very straight forward. Except when it is not. Remember that Bash is particular about white-space. For example:

```
a = 1
```

This will not work. Bash will interpret this a calling the function 'a' with parameters '=' and '1'. This demonstrates one of the dangers of having very flexible and lax syntax. What happens is you end up with very restrictive syntax.

Bash is not strictly typed, which means you do not need to specify the type before defining a variable. A variable can be assigned to anything. Strings need a bit of work, however.

```
a="words with spaces"
```

There cannot be a space between 'a' or '=' or the first ("). This is fairly intuitive. You can use single inverted commas (') as well. There is a difference between using (') and (") however which we will get to later but for simple strings they are interchangeable.

This is where things start to get… weird.

```
a=$(ls)
```

This assigns 'a' to the result of the 'ls' command. If you subsequently echo 'a' you'll see what an 'ls' produces (there are some issues with new lines to contend with however). You will see this $(??) syntax a fair bit so just keep it in mind. We'll cover it in a bit more depth when we discuss Subshells.

For this next part I strongly recommend opening a terminal and just playing around. Consider the following example.

```
a="quack"
echo $a$a              > quackquack
echo "${a}s"           > quacks
echo "${a:1:4}"        > uac
```

Here we have 'a' set to a fairly boring 'quack'. This allows us to just concatenate things all together or take subsets.

Arrays

As seen before, they can be assigned between round brackets with plenty of spaces.

```
a=( first_element second_element third_element )
```

Accessing the values is straightforward too.

```
b=${a[0]}
```

Yes… curly braces. With round braces, Bash would want to run 'a[0]' as a function.
There are a lot of other shortcuts for arrays too.

a[0]=5	Assignment is easy
${arr[@]}	Get all elements
${!arr[@]}	Get array indices
${#arr[@]}	Get the length of the array
a+=(value)	Append value to the end of a
${a[@]:i:j}	Get the values from index i through index j.

Creating Variables in Strange Ways

It is also possible to create variables at strange times during execution.

```
echo ${a_variable:="The contents of a variable"}
echo $a_variable
```

I do not have much to say about this. There are likely specific use-cases for this but in general I would prefer a more 1-line/1-thing way of coding.

Exporting Variables

So, in C++ if we defined a variable in **main** we wouldn't expect to be able to call that variable from another function.

```
int function()
{
      cout << a << endl;     // Compiler says NO!
}
int main()
{
      int a = 1;
      function();
}
```

This is generally how scope works. There are global variables which are accessible by everyone everywhere but as good programmers, global variables are to be avoided where possible. It possible in Bash to subvert this scope using the **export** command. Consider the following two pieces of code.

script1.sh

```
a=5
./script2.sh
```

script2.sh

```
echo $a
```

In any other language, we would expect a crash of some kind. In Bash, as 'a' hasn't been assigned yet it will just ignore it and print out nothing. If we add an **export** call prior to calling script2, 'a' will now work within script2 and will print out '5'.

script1.sh

```
a=5
export a
./script2.sh
```

script2.sh

```
echo $a
```

Again, there may be use cases for this and you should be aware it exists, but I have never found a use for it. It seems to violate some basic 'good programming' rule so use with caution.

Redirection

Up until now we have sort of glossed over input and output. I have mentioned that programs in Bash essentially eat and spew out strings of characters. The question then is, where are they getting these characters from and where are they sending them to. By default, the answer is the terminal. If a program requires input it will usually demand the user type something in. If they require output, they will simply output to the screen.

For output the most obvious examples are either 'echo Hello world' or 'ls'. These output things to the screen. Input is a bit more finicky as the program needs to know when you have finished typing. For some functions, this is easy enough. Using the following command:

```
read a b
```

Then typing some text and pressing enter will read into the variables 'a' and 'b' whatever you typed (try 'echo $a' and 'echo $b' to see). Interestingly because of how read works, a will be the first word, b everything else.

What about 'wc'. 'wc' counts the number of lines, words and characters which means it will accept multiple lines. To exit 'wc' you need to use 'Ctrl-D' which sends the 'End of File' character to the terminal. It should then print out your 'wc'.

Now it is pretty rare to want to type out something to want to find the word count. Word count does take a file as a parameter so you could just do 'wc filename'. Alternatively, you could use redirection.

```
wc < input_file.txt
```

This changes where 'wc' gets its input from so it comes from a file rather than from the terminal (there is a good chance you've already done when running C++ programs).

If you want to save the output of a program, rather than printing to the terminal, you can output to a file.

```
wc > output_file.txt
```

You can also do both:

```
wc < input_file.txt > output_file.txt
```

You can also append to a file (rather than overwriting it) using the '>>' operator instead.

```
wc >> output_file.txt
wc >> output_file.txt
```

This results in two sets of somewhat meaningless numbers in one file (rather than one).

Pipes

Sometimes, instead of reading or writing to a file, you want to bunch lots of commands together. For example, perhaps you need to know the number of files in a directory. Well, using 'ls' will give you all the files in a directory. Using 'wc -l' will give you the number of lines in a block of text. What if you could take the output of 'ls' and use it as the input of 'wc -l'.

```
ls | wc -l
```

This will simply print out the number of files in a directory. Learning to use pipes will massively improve your ability to use Bash as this is really the point of it. There are dozens of little functions like read, cut, paste, ls, wc, cp, mv, grep etc which when piped together can perform all kinds of tasks with ease, that would be otherwise menial.

Here Strings

So, there is also a '<<<' operator. In its most basic form, it simply passes what is on the right to what is on the left as input.

Consider:

```
cat <<< "Hello World"
```

It prints "Hello World"… another print command, just what we need!

Some more examples to test.

```
cat <<< $(ls)
cat <<< $(wc test_input.txt)
```

Not looking particularly useful, but much more useful when used with **read**.

Briefly, **read** can be used thusly:

```
read a b c
```

Once entered, the user must type in a line of text. 'a' will now contain the first word, 'b' the second and 'c' whatever was left over. These can now be accessed as $a, $b and $c like normal variables.

```
read lines words chars filename <<< $(wc test_input.txt)
```

This code reads the output of wc and puts the corresponding values in the corresponding variables.

```
echo $lines      # Prints the number of lines
echo $words      # Prints the number of words
echo $chars      # Prints the number of characters
echo $filename   # Prints the filename
```