Lecture 05: Bash Expansion

Globbing

Bash has a lot of shortcuts for referring to different kinds of files. As you can imagine, without windows it would be otherwise hard to copy 1000 files from one place to another without losing your mind. This comes under the term 'Globbing' which is kind of gluing things into a blob so it is easier to work with them. Learning Globbing shortcuts is quite useful. It can be used for searching/sorting tasks where you are looking for particular files which match a pattern and then doing things with them.

Wild Card (*)

'*' basically means an arbitrary amount of text. Anything that matches an arbitrary amount of text will be considered. There is an example provided (globbing_1.sh) that will explain this better than reading it can but in effect the '*' will expand to whatever is available. Some examples

```
ls *.txt          # Only lists files that end in .txt
ls *a*.txt        # Only lists files that contain an 'a'
rm *              # Deletes everything in the directory
```

There is some subtlety to this. It is not like you can type:

```
touch *.txt
```

Intuitively this might make all the possible text files. However, it does not. Instead, it just updates all the .txt files that are available. What the interpreter is actually doing is replacing the '*' with all valid substitutions. Those substitutions need to already exist.

```
ls *.txt
```

Becomes:

```
ls my_file.txt a.txt some_other_file.txt
```

Assuming those are the three files in the directory.

Wild Card (?)

The '?' character works similarly to the '*' but only works for a single character.

```
ls ?.txt
```

Will list 'a.txt', but not 'aa.txt'.

Other kinds of basic Globbing

There are dozens of single-character wildcards which you can substitute in a string.

[a-f]	A single character in a specified range
[aeiou]	Matches one of these
[[:alpha:]]	Letters only
[[:punct:]]	Punctuation
[[:digit:]]	Digits

Note there is some differences in the number of square braces.

Curly Brackets {}

We can do our 'touch' plan. It just requires some more specific instructions.

```
touch {a..z}.txt
```

This will create files 'a.txt', 'b.txt', 'c.txt' … 'z.txt'.

This syntax also allows commas.

```
touch {file1,file2,file3}.txt
```

Producing: file1.txt, file2.txt, file3.txt.

```
touch {a..z}{a..z}.txt
```

This makes every two-letter combination. aa.txt, ab.txt … zy.txt, zz.txt.

The Exclamation '!'

This kind of expansion really just needs an example:

```
my_var=51
var_name="my_var"
echo ${!var_name}
```

So, what is going on? It starts fairly straightforward with 'my_var' being assigned to the value 51. We then create a variable (which is a string) called 'var_name'. If we wanted to copy the value we would use:

```
var_name=$my_var
```

We did not do this. 'var_name' is now just a string with the value 'my_var'. Now for the hard part. By using the '!' we are turning 'var_name' into its value (which is 'my_var'). This is now equivalent to writing echo $my_var which prints 51! So, what was the point of all this? Sometimes you want to be able to turn a string into a variable name.

Environment Variables

Now this might seem like a sudden change in topic, but unfortunately one of the expansion characters '~' really needs an understanding of environmental variables to make sense.

In Unix, there are some variables that are there by default. Probably the most important of these are:

```
$HOME:     The place you call home… usually /home/username
$PATH:     The list of places where Unix looks when you type
           something
$SHELL:    The shell program you are using (hopefully /bin/bash)
```

These can be seen by just using 'echo $HOME' etc

$Path

Understanding what the Path is will save you a lot of headaches when using Linux. In essence, when you type something like 'ls' into the terminal, what is actually happening is Linux is going… "what is 'ls'? I guess I should try to find it". Linux then looks in all of the locations listed on the path. The path has a very distinct look. It consists of locations separated by ':' symbols (this is sort of like a comma). There will be some useful locations there too, such as '/bin', '/usr/bin'. If you go to these locations or more simply just type 'ls /bin' into the terminal, you will see that they contain a whole series of different functions. For example '/bin' contains 'pwd' and 'ls'.

One function that is not in these locations is 'cd'. It may not be immediately obvious why 'cd' is not on the path, but there is a reason. 'cd' is actually built into the shell program (i.e., the terminal). The reason for this is how programs work. Most programs have their own private universe in which they set variables, perform operations, and calculate. When a program finishes, to prevent memory leaks, all of these things are removed and cease to exist. This means if you were to try to write a 'cd' program, when it had wrapped up, it would reset the terminal to its original state (i.e., it would end where it started). So, it wouldn't work as a 'cd' because the directory wouldn't change.

Many programs (when installed) will request to placed on the path so they can be directly called. This usually consists of a "PATH=????:$PATH" command where ???? is the new location to be added.

Some other Environment Variables

Just for reference we have a few more:

```
$USER      # The current user
$HOSTNAME  # The hostname of the machine the script is running on
$RANDOM    # A random number from 0-32767
$LINENO    # The bash script's current line number
```

Some of these are a bit strange… because really $RANDOM is not what I think of when I think 'environment variable'. It is not even really a variable…

Arithmetic

A very standard part of coding is doing some basic maths. To be honest, the kinds of programs that do a lot of maths probably should not be written in Bash, but quite often you will need to do some.

Using let

We have already seen the **let** expression in **while** loops.

```
let a=2+2
```

Let has strict white-space requirements. These can be overcome by using “”.

```
let “a = 1000 * $1”
```

Double Round Brackets (())

Expressions encapsulated within $(()) are generally fairly easy to parse and have fewer white-space requirements.

```
echo “2 + 2 is $((2 + 2))”
a=2
b=5
echo “a + b is $((a + b))”
echo “a + b is $(($a + $b))”
```

This is a convenient way to do string substitution.

Using expr

Expression is like **let** except it does not do assignment and does not require quotes, it just prints the value

```
expr 2 + 2
```

It can be nested to for assignment.

```
a=$( expr 2 + 2 )
```

Braces

So by now… brackets/braces/parentheses are probably floating around in your head in a garbled mess. So here is a quick set of examples for each that you can refer back to if you ever get completely confused.

()	Used to define arrays a=(A B C) Used to expand commands into text $(ls)
(())	Arithmetic • ((2+2)) For Loops • for ((i=0; i < 20;i++))
[]	Used in if conditions • **if** [a -gt 5] Array access • a[0]=1 • b=${a[0]}
[[]]	A more advanced version of [] for conditions
{}	Expansion: • **touch** {a,b,c}.txt For Loops: • **for** i **in** {0..19}