Bash Scripts

As we've seen so far, a bash script really just seems to be a file with a series of bash commands. Ultimately speaking, any file that is just a series of shell commands is a *shell script*, however, since we're primarily working in bash, we call them *bash scripts*. These scripts are invoked just as any other program (i.e. by providing the shell a full or relative path to the script). Most shells do follow similar, syntax, though, so what works in one shell ought to mostly work in another.

3.1 Running a Bash Script

This was touched on a bit in chapter one, but it's good to go over it again: When you run a script with the **interactive shell** (i.e. the shell you're working in), you're launching an **invoked shell** that is running the script itself. A great example of this is in the example of a file named `changeDirectoryAndPrint` in your home directory:

```
cd /usr/lib
pwd
```

If you then run the following commands (after having changed execution permissions of the script `changeDirectoryAndPrint`), you'll notice something curious:

```
echo In interactive Shell && pwd && echo running script
```

> In interactive Shell /home/mjlny2/2750Materials/module3/ch5 running script

```
./changeDirectoryAndPrint
```

> /usr/lib

```
echo Finished with Script && pwd
```

> Finished with Script /home/mjlny2/2750Materials/module3/ch5

Granted, in the above examples, your `pwd` output may be different because you're likely in a different directory, however, notice that after we ran the script `changeDirectoryAndPrint` when we then ran `pwd`, our path was still in the original directory of the script itself. What happened is that the shell spins off a sub shell that runs the commands, while the interactive shell sits there waiting for the sub shell to finish its commands.

To run the commands themselves within the interactive shell, you'll need to use the `source` command:

```
source ./changeDirectoryAndPrint
pwd
```

> /usr/lib

5.2 Bash Scripts

Up to now, many of the "bash scripts" that we've seen have been relatively uninterseting. A bash script is really just a file that consists of a series of commands to be run in a sub shell (or a shell, if you source the script). We've been relatively lax about how we've been writing these "bash scripts". When writing a script, it is necessary that you prefix your entire script with:

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

By doing so, you ensure that your script is always run by bash, as opposed to any other shell that your terminal might be running. As of now, we've been able to get away with writing our scripts without the prefix because we've specifically been running bash.

Let's revisit `takesArguments.sh` from chapter 3:

```
#!/bin/bash

echo arg0 $0
echo arg1 $1
echo arg2 $2
echo arg3 $3
echo arg4 $4
```

The script `takesArguments.sh` is nothing too terribly exciting, except it is a great example of *postional parameters*. These parameters refer specifically to the positions they were passed in:

```
./takesArguments.sh firstArg secondArg thirdArg fourthArg
```

> arg0 /home/mjlny2/takesArguments.sh arg1 firstArg arg2 secondArg arg3 thirdArg arg4 fourthArg

The first of these arguments, `arg0`, refers entirely to the command itself (by its full name). Naturally, `firstArg` refers to `$1`, `secondArg` to `$2`, and so on.

Note: to execute bash scripts without referring to the relative or full path, you will need to place them in a directory in the command search path (see chapter 3).

5.3 Shell Script Execution

There are a number of things to keep in mind as a shell script executes. Each script runs commands separated by new lines (multiple commands can be on the same line if they're delimited by a `;`). Comments, unlike c-style languages, are started with a `#`.

The general flow of a shell script is:

1. The shell tokenizes the command and the input (all arguments are assigned *positional parameter* values)
2. Read next command in sequence
    - If no commands next in sequence, **return exit status** (typically 0 for success)
3. Runs next command.
    - If succeeds, return to #2
4. Check if command is build-in:
    - If command not built in: return to #2
    - If command is built in: **return exit status** (typically non-zero for failure)

As we move throughout the chapter, we may come across the terms:

- Commandlist: commands in a shell script separated by new lines or semicolons
- Wordlist: words delimited by whitespace.

5.4 Positional Parameters

In section 5.2, we saw positional parameters, and in section 5.3 we saw that the shell tokenizes the command line input to refer to those positional parameters. The one thing we have not seen out of this is that it is possible to obtain all arguments passed in with either the `*` or `@` symbols. Create a file: `takesArgumentsAgain.sh`:

```
#!/bin/bash

echo arg0 $0
echo arg1 $1
echo arg2 $2
echo arg@ $@
echo arg* $*
```

And make sure to change the mode so that you can execute the file:

```
./takesArgumentsAgain.sh one two three four
```

> arg0 ./takesArgumentsAgain.sh arg1 one arg2 two arg@ one two three four arg* one two three four

Both the `@` and the `*` act as wild cards so that you can grab all of the arguments. This could come in handy if there were significantly more than 2 or 3 arguments, and you needed an array to be able to work with them all.