Learn Bash The Hard Way

An introduction to Bash using 'The Hard Way' method.

Chapter 1. Core Git

This bash course has been written to help bash users to get to a deeper understanding and proficiency in bash. It doesn't aim to make you an expert immediately, but you will be more confident about using bash for more tasks than just one-off commands.

1.1. Why learn bash?

There are a few reasons to learn bash in a structured way:

- Bash is ubiquitous
- Bash is powerful

You often use bash without realising it, and people often get confused by why things don't work, or how things work when they're given one-liners to run.

It doesn't take long to get a better understanding of bash, and once you have the basics, its power and ubiquity mean that you can be useful in all sorts of contexts.

1.2. What You Will Get

This course aims to give students:

- A hands-on, quick and practical understanding of bash
- Enough information to understand what is going on as they go deeper into bash
- A familiarity with advanced bash usage

It does not:

- Give you a mastery of all the tools you might use on the command line, eg sed, awk, perl
- Give a complete theoretical understanding of all the
- Explain everything.

You are going to have to think sometimes to understand what is happening. This is the Hard Way, and it's the only way to really learn. This course will save you time as you scratch your head later wondering what something means, or why that StackOverflow answer worked.

Sometimes the course will go into other areas closely associated with bash, but not directly bash-related, eg specific tools, terminal knowledge. Again, this is always oriented around my decades of experience using bash and other shells.

1.3. Assumptions

It assumes some familiarity with *very* basic shell usage and commands. For those looking to get to that point, I recommend following this set of mini-tutorials:

https://learnpythonthehardway.org/book/appendixa.html

It also assumes you are equipped with a bash shell and a terminal. If you're unsure whether you're in bash, type:

```
echo $BASH_VERSION
```

into your terminal. If you get a bash version string output then you in bash.

1.4. How The Course Works

The course *demands* that you type out all the exercises to follow it.

Frequently, the output will not be shown in the text, or even described.

Explanatory text will assume you typed it out.

This is really important: you must get used to working in bash, and figuring out what's going on by scratching your head and trying to work it out before I explain it to you.

Each section is self-contained, and must be followed in full. To help show you where you are, the shell commands are numbered 1-n and the number is followed by a '$' sign, eg:

```
1$ #first command
2$ #second command
```

At the end of each section is a set of 'cleanup' commands (where needed) if you want to use them.

1.5. Structure

This book is structured into four parts:

1.5.1. Core Bash

Core foundational concepts essential for understanding bash on the command line.

1.5.2. Scripting Bash

Gets your bash scripting skills up to a proficient point.

1.5.3. Tips

A primer on commonly-used techniques and features that are useful to know about.

1.5.4. Advanced Bash

Building on the first three chapters, and introducing some more advanced features, this chapters

takes your bash skills beyond the norm.

1.6. Introduction

1.6.1. What is bash?

Bash is a shell program.

A shell program is typically an executable binary that takes commands that you type and (once you hit return), translates those commands into (ultimately) system calls to the Operating System API.

1.6.2. Other shells

Other shells include:

- sh
- ash
- dash
- ksh
- csh
- tclsh

These other shells have different rules, conventions, logic, and histories that means they can look similar.

Because other shells are also programs, they can be run from in each other!

Here you run csh from within your bash terminal. Note that you get a different prompt (by default):

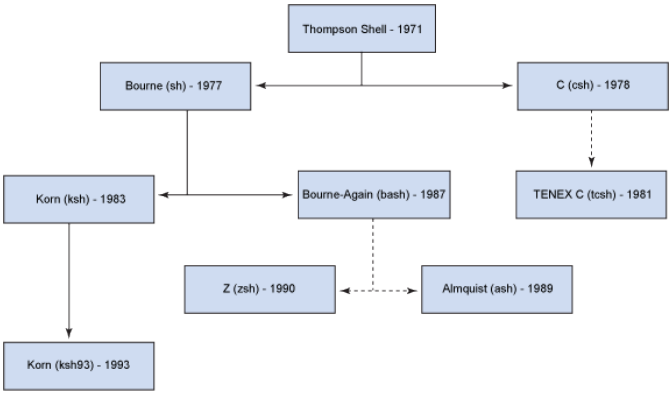
```
1$ csh
2% echo $dirstack
```

Typically, a csh will give you a prompt with a percent sign, while bash will give you a prompt with a dollar sign. This is configurable, though, so your setup may be different.

The 'dirstack' variable is set by csh and will output something meaningful. It’s not there by default in bash (try it!)

1.6.3. History of bash

This diagram helps give a picture of the history of bash:



Bash is called the 'Bourne Again SHell'. It is a descendant of the Thompson Shell and then the Bourne 'sh' shell. Bash has other 'siblings' (eg ksh), 'cousins' (eg tcsh), and 'children', eg 'zsh'.

The details aren't important, but it's important to know that different shells exist and they can be related and somewhat compatible.

Bash is the most widely seen and used shell as of 2017. However, it is still not uncommon to end up on servers that do not have bash!

1.6.4. What You Learned

- What a shell is
- How to start up a different shell
- The family tree of shells

1.6.5. What Next?

Next you look at two thorny but ever-present subjects in bash: globbing and quoting.

1.6.6. Exercises

1) Run 'sh' from a bash command line. What happens?

2) What commands can you find that are work in 'bash', but do not work in 'sh'?

1.7. Unpicking the shell: Globbing and Quoting

You may have wondered what the '*' in bash commands really means, and how it is different from regular expressions. This section will explain all, and introduce you to the joy of quoting in bash.

1.7.1. Globbing

Type these commands into your terminal

```
1$ mkdir lbthw_tmp           # Line 1
2$ cd lbthw_tmp              # Line 2
3$ touch file1 file2 file3   # Line 3
4$ ls *                      # Line 4
5$ echo *                    # Line 5
```

- Line 1 above makes a new folder that should not exist already.
- Line 2 moves into that folder.
- Line 3 creates three files (file1,2,3).
- Line 4 runs the 'ls' command, which lists files, asking to list the files matching '*'
- Line 5 runs the echo command using '*' as the argument to echo

What you should have seen was the three files listed in both cases.

The shell has taken your '' **character and converted it to match all the files in the current working directory. In other words, it's converted the** '' character into the string "file1 file2 file3" and then processed the resulting command.

1.7.2. Quoting

What do you think will be output happen if we run these commands?

Think about it first, make a prediction, and then type it out!

```
6$ ls '*'         # Line 1
7$ ls "*"         # Line 2
8$ echo '*'       # Line 3
9$ echo "*"       # Line 4
```

- Line 1 lists files matching the '*' character in single quotes
- Line 2 lists files matching the '*' character in double quotes
- Line 3 'echo's the '*' character in single quotes
- Line 4 'echo's the '*' character in double quotes

This is difficult even if you are an expert in bash!

Was the output what you expected? Can you explain it? Ironically it may be harder to explain if you have experience of quoting variables in bash!

Quoting in bash is a very tricky topic. You may want to take from this that quoting globs removes their effect. But in other contexts single and double quotes have different meanings.

Quoting changes the way bash can read the line, making it decide whether to take these characters and transform them into something else, or just leave them be.

What you should take from this is that 'quoting in bash is tricky' and be prepared for some head-scratching later!

1.7.3. Other glob characters

'*' is not the only globbing primitive. Other globbing primitives are:

- ? - matches any single character
- [abd] - matches any character from a, b or d
- [a-d] - matches any character from a, b, c or d

Try running these commands and see if the output is what you expect:

```
10$ ls *1             # Line 1
11$ ls file[a-z]      # Line 2
12$ ls file[0-9]      # Line 3
```

- Line 1 list all the files that end in '1'
- Line 2 list all files that start with 'file' and end with a character from a to z
- Line 3 list all files that start with 'file' and end with a character from 0 to 9

Differences with Regular Expressions

While globs look similar to regular expressions (regexes), they are used in different contexts and are separate things.

The '*' characters in this command:

```
13$ rename -n 's/(.*)(.*)/new$1$2/' *      # Line 1
'file1' would be renamed to 'newfile1'   # Line 2
'file2' would be renamed to 'newfile2'   # Line 3
'file3' would be renamed to 'newfile3'   # Line 4
```

- Line 1 prints the files that would be renamed by the rename command if the -n flag were removed
- Lines2-4 show the files that would be renamed

have a different significance depending on whether it is being treated as a glob or a regular expression.

> This assumes you have the program 'rename' installed.

Again, the key takeaway here is that context is key.

Note that '.' has no meaning as a glob, and that some shells offer extended globbing capabilities.

Bash offers 'extended globbing', which we do not cover here.

Cleanup

Now clean up what you just did:

```
14$ cd ..
15$ rm -rf lbthw_tmp
```

1.7.4. What You Learned

- What a glob is
- Globs and regexes are different
- Single and double quotes around globs can be significant!

1.7.5. What Next?

Next up is another fundamental topic: variables.

1.7.6. Exercises

1) Create a folder with files with very similar names and use globs to list one and not the other.

2) Research regular expressions online.

3) Research the program 'grep'. If you already know it, read the grep man page. (Type 'man grep').

1.8. Variables in Bash

As in any programming environment, variables are critical to an understanding of bash. In this section you'll learn about variables in bash and some of their subtleties.

1.8.1. Basic Variables

Start by creating a variable and echoing it.

```
1$ MYSTRING=astring
2$ echo $MYSTRING
```

Simple enough: you create a variable by stating its name, immediately adding an equals sign, and then immediately stating the value.

Variables don't need to be capitalised, but they generally are by convention.

To get the value out of the variable, you have to use the dollar sign to tell bash that you want the variable dereferenced.

1.8.2. Variables and Quoting

Things get more interesting when you start quoting.

Quoting used to group different 'words' into a variable value:

```
3$ MYSENTENCE=A sentence
4$ MYSENTENCE="A sentence"
5$ echo $MYSENTENCE
```

Since (by default) the shell reads each word in separated by a space, it thinks the word 'sentence' is not related to the variable assigment, and treats it as a program. To get the sentence into the variable with the space is in it, you can enclose it in the double quotes, as above.

Things get even more interesting when we embed other variables in the quoted string:

```
6$ MYSENTENCE="A sentence with $MYSTRING in it"
7$ MYSENTENCE='A sentence with $MYSTRING in it'
```

If you were expecting similar behaviour to the previous section you may have got a surprise!

This illustrated an important point if you're reading shell scripts: the bash shell translates the variable into its value if it's in double quotes, but does not if it's in single quotes.

Remember from the previous section that this is not true when globbing! Globs are not expanded when in either single or double quotes. Confusing isn't it?

1.8.3. Shell Variables

Some variables are special, and set up when bash starts:

```
8$ echo $PPID # Line 1
9$ PPID=nonsense # Line 2
10$ echo $PPID # Line 3
```

- Line 1 - PPID is a special variable set by the bash shell. It contains the bash's parent process id.
- Line 2 - Try and set the PPID variable to something else.
- Line 3 - Output PPID again.

What happened there?

If you want to make a readonly variable, put 'readonly' in front of it, like this:

```
11$ readonly MYVAR=astring
12$ MYVAR=anotherstring
```

1.8.4. env

Wherever you are, you can see the variables that are set by running this:

```
13$ env
TERM_PROGRAM=Apple_Terminal
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000000
TMPDIR=/var/folders/mt/mrfvc55j5mg73dxm9jd3n4680000gn/T/
PERL5LIB=/home/imiell/perl5/lib/perl5
GOBIN=/space/go/bin
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.2BE31oXVrF/Render
TERM_PROGRAM_VERSION=361.1
PERL_MB_OPT=--install_base "/home/imiell/perl5"
TERM_SESSION_ID=07101F8B-1F4C-42F4-8EFF-1E8003E8A024
HISTFILESIZE=1000000
USER=imiell
SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.uNwbe2XukJ/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0x0:0x0
PATH=/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/Users/imiell/google-cloud-
sdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/space/git/shutit:/s
pace/git/work/bin:/space/git/home/bin:~/.dotfiles/bin:/space/go/bin
PWD=/space/git/work
LANG=en_GB.UTF-8
XPC_FLAGS=0x0
HISTCONTROL=ignoredups:ignorespace
XPC_SERVICE_NAME=0
HOME=/Users/imiell
SHLVL=2
PERL_LOCAL_LIB_ROOT=/home/imiell/perl5
LOGNAME=imiell
GOPATH=/space/go
DISPLAY=/private/tmp/com.apple.launchd.lwUJWwBy9y/org.macosforge.xquartz:0
SECURITYSESSIONID=186a7
PERL_MM_OPT=INSTALL_BASE=/home/imiell/perl5
HISTTIMEFORMAT=%d/%m/%y %T
HISTFILE=/home/imiell/.bash_history
_=/usr/bin/env
OLDPWD=/Users/imiell/Downloads
```

The output will be different wherever you run it.

1.8.5. export

Type in these commands, and try to predict what will happen:

```
14$ MYSTRING=astring
15$ bash
16$ echo $MYSTRING
17$ exit
18$ echo $MYSTRING
19$ unset MYSTRING
20$ echo $MYSTRING
21$ export MYSTRING=anotherstring
22$ bash
23$ echo $MYSTRING
24$ exit
```

Based on this, what do you think export does?

You've already seen that a variable set in a bash terminal can be referenced later by using the dollar sign.

But what happens when you set a variable, and then start up another process?

In this case, you set a variable (MYSTRING) to the value "astring", and then start up a new bash shell process. Within that bash shell process, MYSTRING does not exist, so an error is thrown. In other words, the variable was not inherited by the bash process you just started.

After exiting that bash session, and unsetting the MYSTRING variable to ensure it's gone, you set it again, but this time 'export' the variable, so that any processes started by the running shell will have it in their environment. You show this by starting up another bash shell, and it 'echo's the new value "anotherstring" to the terminal.

It's not just shells that have environment variables! All processes have environment variables.

1.8.6. Arrays

Worth mentioning here also are arrays. One such built-in, read only array is BASH_VERSINFO. As in other languages, arrays in bash are zero-indexed.

Type out the following commands, which illustrate how to reference the version information's major number:

```
25$ bash --version
26$ echo $BASH_VERSINFO
27$ echo $BASH_VERSINFO[0]
28$ echo ${BASH_VERSINFO[0]}
29$ echo ${BASH_VERSINFO}
```

Arrays can be tricky to deal with, and bash doesn't give you much help!

The first thing to notice is that if the array will output the item at the first element (0) if no index is given.

The second thing to notice is that simply adding [0] to a normal array reference does not work. Bash treats the square bracket as a character not associated with the variable and appends it to the end of the array.

You have to tell bash to treat the whole string "BASH_VERSION[0]" as the variable to be dereferenced. You do this by using the curly braces.

These curly braces can be used on simple variables too:

```
30$ echo $BASH_VERSION_and_some_string
31$ echo ${BASH_VERSION}_and_some_string
```

In fact, 'simple variables' can be treated as arrays with one element!

```
32$ echo ${BASH_VERSION[0]}
```

So all bash variables are 'really' arrays!

Bash has 6 items (0-5) in its BASH_VERSINFO array:

```
33$ echo ${BASH_VERSINFO[1]}
34$ echo ${BASH_VERSINFO[2]}
35$ echo ${BASH_VERSINFO[3]}
36$ echo ${BASH_VERSINFO[4]}
37$ echo ${BASH_VERSINFO[5]}
38$ echo ${BASH_VERSINFO[6]}
```

As ever with variables, if the item does not exist then the output will be an empty line.

1.8.7. Associative Arrays

Bash also supports 'associative arrays', but only in versions 4+.

To check whether you have version 4+, run:

```
39$ bash --version
```

With associative arrays, you use a string instead of a number to reference the value:

```
40$ declare -A MYAA=([one]=1 [two]=2 [three]=3)
41$ MYAA[one]="1"
42$ MYAA[two]="2"
43$ echo $MYAA
44$ echo ${MYAA[one]}
45$ MYAA[one]="1"
46$ WANT=two
47$ echo ${MYAA[$WANT]}
```

As well as not being compatible with versions less than 4, associative arrays are quite fiddly to create and use, so I don't see them very often.

1.8.8. What You Learned

- Basic variable usage in bash
- Variables and quoting
- Variables set up by bash
- env and export
- Bash arrays

1.8.9. What Next?

Next you will learn about another core language feature implemented in bash: functions.

1.8.10. Exercises

1) Take the output of 'env' in your shell and work out why each item is there and what it might be used by. You may want to use 'man bash', or use google to figure it out. Or you could try re-setting it and see what happens.

2) Find out what the items in BASH_VERSINFO mean.

1.9. Functions in Bash

From one angle, bash can be viewed as a programming language, albeit a quite slow and primitive one.

One of the language features it has are the capability to create and call functions.

This leads us onto the topic of what a 'command' can be in bash: function, alias, program or 'builtin'.

1.9.1. Basic Functions

Start by creating a simple function

```
1$ function myfunc {
    echo Hello World
}
2$ myfunc
```

By declaring a function, and placing the block of code that needs to run inside curly braces, you can then call that function on the command line as though it were a program.

1.9.2. Arguments

Unlike other languages there is no checking of functions' arguments.

Predict the output of this, and then run it:

```
3$ function myfunc {
    echo $1
    echo $2
}
4$ myfunc "Hello World"
5$ myfunc Hello World
```

Can you explain the output?

Arguments to functions are numbered, from 1 to n. It’s up to the function to manage these arguments.

1.9.3. Variable Scope

Variables can have scope in bash. This is particularly useful in functions, where you don’t want your variables to be accessible from outside the function.

These commands illustrate this:

```
6$ function myfunc {
    echo $myvar
}
7$ myfunc
8$ myvar="Hi from outside the function"
9$ myfunc
```

Bash functions have no special scope. variables outside are visible to it.

There is, however, the capability within bash to declare a variable as local:

```
10$ function myfunc {
    local myvar="Hi from inside the function"
    echo $myvar
}
11$ myfunc
12$ echo $myvar
13$ local myvar="Will this work?"
```

The variable declared with 'local' is only viewed and accessed within the function, and doesn't interfere with the outside.

'local' is an example of a bash 'builtin'. Now is a good time to talk about the different types of commands.

1.9.4. Functions, Builtins, Aliases and Programs

Now is a good time to point out another area of bash which can cause confusion.

There are at least 4 ways to call commands in bash:

- Builtins
- Functions
- Programs
- Aliases

Let's take each one in turn.

Builtins

Builtins are commands that come 'built in' to the bash shell program. Normally you can't easily tell the difference between a builtin, a program or a function, but after reading this you will be able to.

Two such builtins are the familiar 'cd' and one called 'builtin'!

```
14$ builtin cd /tmp
15$ cd -
16$ builtin grep
17$ builtin notaprogram
```

As you've probably guessed from typing the above in, the 'builtin' builtin calls the builtin program (this time 'cd'), and throws an error if no builtin exists

In case you didn't know 'cd -' returns you to the previous directory you were in.

Functions

Functions we have covered above, but what happens if we write a function that clashes with a

builtin? What if you create a function called 'cd'?

```
18$ function cd() {
    echo 'No!'
}
19$ cd /tmp
20$ builtin cd /tmp
21$ cd -
22$ unset -f cd
23$ cd /tmp
24$ cd -
```

At the end there you 'unset' the function 'cd'. You can also 'unset -v' a variable. Or just leave the -v out, as it will assume you mean a variable by default.

```
25$ declare -f
26$ declare -F
```

If you want to know what functions are set in your environment, run 'declare -f'. This will output the functions and their bodies, so if you just want the names, use the '-F' flag.

Programs

Programs are executable files. Commonly-used examples of these are programs such as 'grep', 'sed', 'vi' and so on.

How do you tell whether a command is a

First, see whether it's a builtin by running 'builtin <command>' as you did before. Then you can also run the 'which' command to determine where the file is on your filesystem.

```
27$ which grep
28$ which cd
29$ which builtin
30$ which doesnotexist
```

Is 'which' a builtin or a program?

Aliases

Finally there are aliases. Aliases are strings that the shell takes and translates to whatever that string is aliased to.

Try this and explain what is going on as you go:

```
31$ alias cd=doesnotexist
32$ alias
33$ cd
34$ unalias cd
35$ cd /tmp
36$ cd -
37$ alias
```

And yes, you can alias alias.

1.9.5. What You Learned

- Basic function creation in bash
- Functions and variable scope
- Differences between functions, builtins, aliases and programs

1.9.6. What Next?

Next you will learn about pipes and redirects in bash. Once learned, you will have all you need to get to writing shell scripts in earnest.

1.9.7. Exercises

1) Run 'typeset -f'. Find out how this relates to 'declare -f' by looking at the bash man page ('man bash').

2) alias alias, override cd. Try and break things. Have fun. If you get stuck, close down your terminal, or exit your bash shell (if you haven't overridden exit!).

1.10. Pipes and redirects

Pipes are redirects are used very frequently in bash. This can cause a problem in that they are used so often by all users of bash that many don't understand their subtleties or how their full power.

This section will lay a firm foundation for you to understand these concepts as we move onto deeper bash topics.

1.10.1. Basic Redirects

Start off by creating a file:

```
1$ mkdir lbthw_pipes_redirects
2$ cd lbthw_pipes_redirects
3$ echo "contents of file1" > file1
```

1.10.2. Basic pipes

Type this in:

```
4$ cat file1 | grep -c file
```

If you don't know what grep is, you will need to learn. This is a good place to start: https://en.wikipedia.org/wiki/Grep

Normally you'd run a grep with the filename as the last argument, but instead here we 'pipe' the contents of file into the grep command by using the 'pipe' operator: '|'.

A pipe takes the standard output of one command and passes it as the input to another. What, then is standard output, really? You will find out soon!

```
5$ cat file2
```

What was the output of that?

Now run this, and try and guess the result before you run it:

```
6$ cat file2 | grep -c file
```

Was that what you expected? If it was, you can explain to the rest of the class :)

If it wasn't, then the answer is related to standard output and other kinds of output.

1.10.3. Standard output vs standard error

In addition to 'standard output', there is also a 'standard error' channel. When you pass a non-existent file to the cat command, it throws an error message out to the terminal. Although the message looks the same as the contents of a file, it is in fact sent to a different output channel. In this case it's 'standard error' rather than 'standard output'.

As a result, it is NOT passed through the pipe to the grep command, and grep counts 0 matches in its output.

To the viewer of the terminal, there is no difference, but to bash there is all the difference in the world!

There is a simpler way to refer to these channels. A number is assigned to each of them by the operating system.

These are the numbered 'file descriptors', and the first three are assigned to the numbers 0,1 and 2.

- 0 is 'standard input'

- 1 is 'standard output'
- 2 is 'standard error'

When you redirect standard output to a file, you use the redirection operator '>'. Implicitly, you are using the '1' file descriptor.

Type this to see an example of redirecting '2', which is 'standard error'.

```
7$ command_does_not_exist
8$ command_does_not_exist 2> /dev/null
```

In the second line above 2 ('standard error') is directed to a file called '/dev/null'.

'/dev/null' is a special file, created by UNIX kernels. It is effectively a black hole into which data can be pumped: anything written to it will be absorbed and ignored.

Another commonly seen redirection operator is '2>&1'.

```
9$ command_does_not_exist 2>&1
```

What this does is tell the shell to send the output on standard error (2) to whatever standard output is pointed at at that point in the command.

Since standard output is pointed at the terminal at that time, standard error is pointed at the terminal. From your point of view you see no difference, since by default . But when we try and redirect to files things get interesting.

Now type these in and try and figure out why they produce different output:

```
10$ command_does_not_exist 2>&1 > outfile
11$ command_does_not_exist > outfile 2>&1
```

This is where things get tricky and you need to think carefully!

Remember that the redirection operator 2>&1 points (standard error) at whatever 1 (standard output) was pointed to at the time. If you read the first line carefully, at the point 2>&1 was used, standard output was pointed at the terminal. So standard error is pointed at the terminal. After that point, standard output is redirected (with the '>' operator) to the file 'outfile'.

So at the end of all this, the standard error of the output of the command 'command_does_not_exist' points at the terminal, and the standard output points at the file 'outfile'.

In the second line, what is different?

The order of redirections is changed. Now:

- the standard output of the command 'command_does_not_exist' is pointed at the file 'outfile'

- the redirection operator 2>&1 points 2 (standard error) to whatever 1 (standard output) is pointed at

So in effect, both standard out and standard error are pointed at the same file (outfile).

This pattern of sending all the output to a single file is seen very often, and few understand why it has to be in that order. Once you understand, you will never pause to think about which way round the operators should go again!

1.10.4. Difference between pipes and redirects

To recap:

- A pipe passes 'standard output' as the 'standard input' to another command
- A redirect sends a channel of output to a file

A couple of other commonly used operators are worth mentioning here:

```
12$ grep -c file < file1
```

The '<' operator redirects standard input to the command from a file, in this case just as 'cat file1 | grep -c file' did.

```
13$ echo line1 > file3
14$ echo line2 > file3
15$ echo line3 >> file3
16$ cat file3
```

The first two lines above use the '>' operator, while the third one uses the '>>' operator. The '>' operator effectively creates the file anew whether it already exists or not. The '>>' operator, by contrast, appends to the end of the file. As a result, only line2 and line3 are added to file3.

Cleanup

Now clean up what you just did:

```
17$ cd ..
18$ rm -rf lbthw_pipes_redirects
```

1.10.5. What You Learned

- File redirection
- Pipes
- The differences between standard output and standard error
- How to redirect one to another

- How to redirect either to a file

1.10.6. What Next?

You're nearly at the end of the first section. Next you will learn about creating shell scripts, and what happens when bash starts up.

1.10.7. Exercises

1) Try a few different commands and work out what output goes to standard output and what output goes to standard error. Try triggering errors by misusing programs.

2) Write commands to redirect standard output to file descriptor '3'.

1.11. Scripts and Startups

This section considers two related subjects: shell scripts, and what happens when the shell is started up.

You've probably come across shell scripts, which won't take long to cover, but shell startup is a useful but tricky topic that catches most people out at some point, and is worth understanding well.

1.11.1. Shell scripts

```
1$ mkdir -p lbthw_scripts_and_startups
2$ cd lbthw_scripts_and_startups
```

A shell script is simply a collection of shell commands that can be run non-interactively. These can be quick one-off scripts that you run, or very complex programs.

The Shebang

Run this:

```
4$ echo '#!/bin/bash' > simple_script
3$ echo 'echo I am a script' >> simple_script
```

You have just created a file called 'simple_script' that has two lines in it. The first consists of two special characters: the hash and the exclamation mark. This is often called 'shebang', or 'hashbang' to make it easier to say. When the operating system is given a file to run as a program, if it sees those two characters at the start, it knows that the file is to be run under the control of another program (or 'interpreter' as it is often called).

Now try running it:

```
5$ ./simple_script
```

That should have failed. Before we explain why, let's understand the command.

The './' characters at the start of the above command tells the shell that you want to run this file from within the context of the current working directory. It's followed by the filename to run.

Similarly, the '../' characters indicate that you want to run from the directory above the current working directory.

This:

```
6$ mkdir tmp
7$ cd tmp
8$ ../simple_script
9$ cd ..
10$ rm -rf tmp
```

will give you the same output as before.

The Executable Flag

That script will have failed because the file was not marked as executable, so you will have got an error saying permission was denied.

To correct this, run:

```
11$ chmod +x simple_script
12$ ./simple_script
```

The chmod program changes the

> The subject of file permissions and ownership can get complex and is not covered in full here. 'man chmod' is a good place to start if you are interested.

The PATH variable

What happens if you don't specify the './' and just run:

```
13$ simple_script
```

The truth I won't know what happens. Either you'll get an error saying it can't find it, or it will work as before.

The reason I don't know is that it depends on how your PATH variable is set up in your environment.

If you run this you will see your PATH variable:

```
$14 echo $PATH
```

Your output may vary. For example, mine is:

```
/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/Users/imiell/google-cloud-
sdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/space/git/shutit:/s
pace/git/work/bin:/space/git/home/bin:~/.dotfiles/bin:/space/go/bin
```

The PATH variable is a set of directories, separated by colons. It could be as simple as:

```
/usr/sbin:/usr/bin
```

for example.

These are the directories bash looks through to find commands, in order.

So what changes the PATH variable? The answer is: bash startup scripts.

But before we discuss them, how can we make sure that simple_script can be run without using './' at the front?

```
15$ PATH=${PATH}:.
16$ simple_script
```

That's how! In the first line you set the PATH to itself, plus the current working directory. It then looks through all the directories that were previously set in your PATH variable, and then finally tries the '.', or local folder, as we saw before.

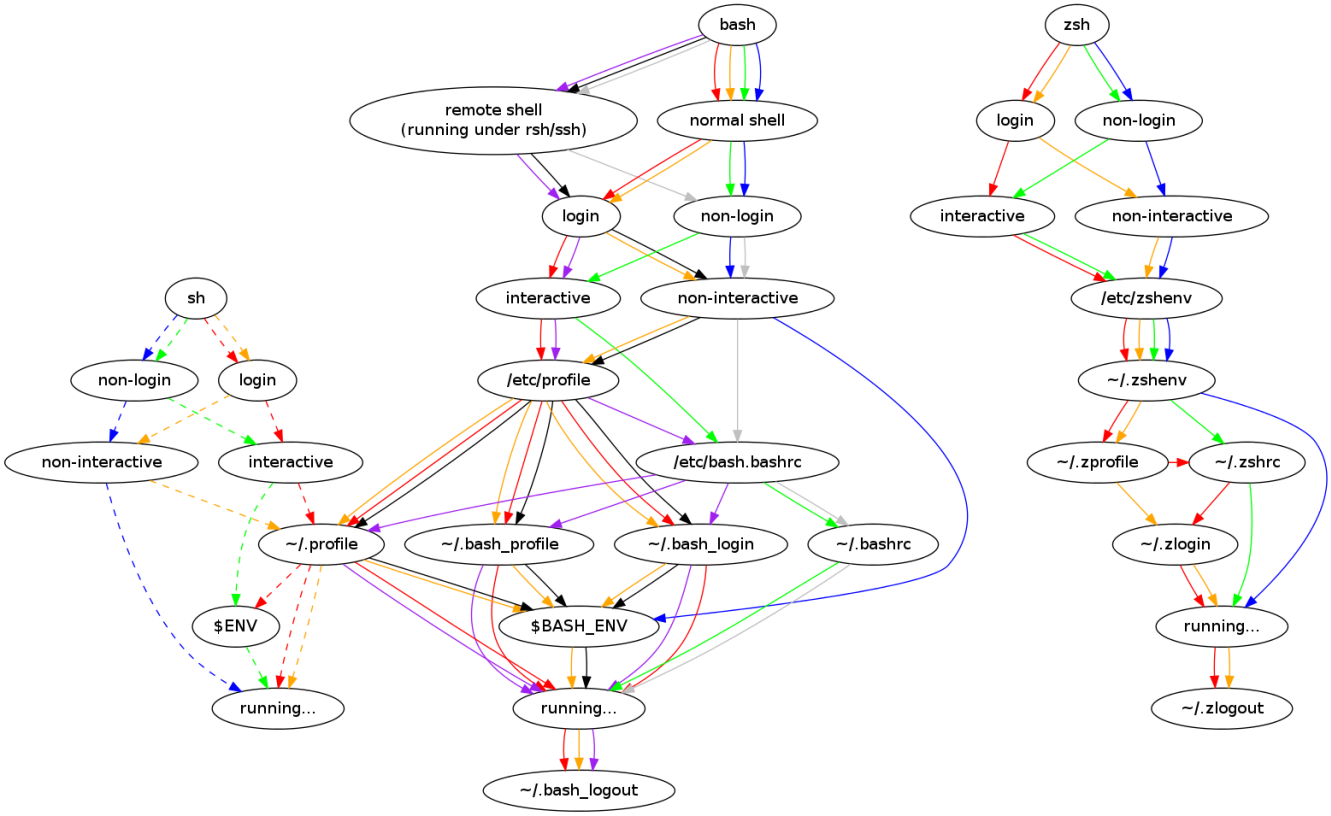
1.11.2. Startup Scripts

Understanding startup scripts and environment variables are key to a lot of issues that you can end up spending a lot of time debugging! If something works in one environment and not in another, the answer is often a difference in startup scripts and how they set up an environment.

1.11.3. Startup Explained

When bash starts up, it calls a runs a series of files to set up the environment you arrive at at the terminal. If you've ever noticed that bash can 'pause' before giving you a prompt, it may be because the startup script

Have a look at this diagram:



Yes, this can be confusing. The diagram shows the startup script order for different shells in different contexts. We are going to follow (from the top) the path from the 'bash' bubble, and ignore the zsh and sh , but it's interesting to note they have their own separate paths (in the case of zsh) and join up at points (in the case of sh and bash).

At each point in this graph the shell you choose either makes a decision about which path to follow, or runs a script.

1.11.4. When You Run Bash

So which path does it take when you run 'bash' on the command line? You're going to follow the graph through here.

The first decision you need to make is whether bash is running 'normally' or as a 'remote' shell. Obviously, you ran bash on a terminal, so it's 'normal'.

From there, you decide if this is a 'login' or a 'non-login' shell. You did not login when you ran bash, so follow 'non-login'.

The final decision is whether bash is running interactively (ie can you type things in, or is bash running a script?). You are on an interactive shell, so follow 'interactive'.

Now, whichever colour line you have followed up to this point, continue with: those files are the ones that get run when bash is started up.

If the file does not exist, it is simply ignored.

Beware

To *further* complicate things, these scripts can be made to call each other in ways that confuse things if you simply believe that diagram. So be careful!

source

Now that you understand builtins, shell scripts, and environments, it's a good time to introduce another builtin: 'source'.

```
17$ MYVAR=Hello
18$ echo 'echo $MYVAR' > simple_echo
19$ chmod +x simple_echo
20$ ./simple_echo
21$ source simple_echo
```

I'm sure you can figure out from that that source runs the script from within

> Most shell scripts have a '.sh' suffix, but this is not required - the OS does not care or take any notice of the suffix.

Cleanup

Now clean up what you just did:

```
22$ cd ..
23$ rm -rf lbthw_scripts_and_startups
24$ unset MYVAR
```

1.11.5. What You Learned

- What the 'shebang' is
- How to create and run a shell script
- The significance of the PATH environment variable
- What happens when bash starts up
- What the builtin 'source' does

1.11.6. What Next?

Well done! You've now finished the first part of the course.

You now have a good grounding to learn slightly more advanced bash scripting, which you will cover in part two.

1.11.7. Exercises

1) Go through all the scripts that you bash session went through. Read through them and try and understand what they're doing. If you don't understand parts of them, try and figure out what's going on by reading 'man bash'.

2) Go through the other files in that diagram that exist on your machine. Do as per 1).

Chapter 2. Scripting

```
some examples of using this
```

```
prefer [[
//-a conditions etc
string vs number comparisons
only example of typing in bash?
arithmetic substitution
```

```
for (( expr1 ; expr2 ; expr3 )) ; do list ; done
select?
```

```
/bin/true / false
pipelines and exit codes
special parameters
```

```
When to use
logging example?
redirection example?
```

2.1. Command Substitution and Evaluation

When writing bash scripts you often want to take the standard output of one command and 'drop' it into the script as though you had written that into it.

This can be achieved with command subsitution.

2.1.1. Command Subsitution Example

An example may help illustrate. Type these commands:

```
1$ hostname
2$ echo 'My hostname is: $(hostname)'
3$ echo "My hostname is: $(hostname)"
```

If that is placed in a script, it will output the hostname of the script it is running on. This can make your script much more dynamic. You can set variables based on the output of commands, add debug.

You will have noticed that if wrapped in single quotes, the special meaning of the '$' sign is ignored

again!

The Two Command Substitution Methods

There are two ways to do command substitution:

```
4$ echo "My hostname is: `hostname`"
5$ echo "My hostname is: $(hostname)"
```

These give the same output and the backticks perform the same function. So which should you use?

Type this:

```
6$ mkdir lbthw_tmp
7$ cd lbthw_tmp
8$ echo $(touch $(ls ..))
9$ cd ..
10$ rm -rf lbthw_tmp
```

What happened there?

You created a folder and moved into it.

The next line is easiest to read from the innermost parentheses outwards.

The 'ls ..' command is run in the innermost parentheses. This outputs the contents of the parent directory.

This output is fed into the next level, where a 'touch' command is run. So the touch command creates a set of empty files, based on the list of the parent directory's contents.

The echo command takes the output of the 'touch' command, which is the list of all the files created.

So, in summary, the line outputs the list of files of the parent directory, and those filenames are also created locally as empty files.

This is an example of how subcommands can be nested. As you can see, the nesting is simple - just place a command wrapped inside a '$()' inside another command wrapped inside a '$()'.

Type this out:

```
11$ rm -rf lbthw_tmp
12$ mkdir lbthw_tmp
13$ cd lbthw_tmp
14$ echo `touch \`ls ..\``
15$ cd ..
```

To nest the backtick version, you have to 'escape' the inner backtick with a backslash, so bash

knows which level the backtick should be interpreted at.

For historical reasons, the backtick form is still very popular, but I prefer the '$()' form because of the nesting benefit. You need to be aware of both, though, if you are looking at others' code!

If you want to see how messy things can get, compare these two lines:

```
16$ echo `echo \`echo \\\`echo inside\\\`\``
17$ echo $(echo $(echo $(echo inside)))
```

and consider which one is easier to read (and write)!

2.1.2. Cleanup

Remove the left-over directory:

```
18$ rm -rf lbthw_tmp
```

2.1.3. What You Learned

- What command substitution is
- How it relates to quoting
- The two command substitution methods
- Why one method is generally preferred over the other

2.1.4. What Next?

Next you will cover tests, which allow you to use what you've learned so far to make your bash code conditional in a flexible and dynamic way.

2.1.5. Exercises

1) Try various command subsitution commands, and plug in variables and quotes to see what happens. 2) Explain why three backslashes are required in the last

2.2. Tests

Tests are a fundamental part of bash scripting, whether it's on the command line in one-liners or much larger scripts.

The subject is very fiddly and confusing. In this section I'll show you some pitfalls, and give rules of thumb for practical bash usage.

2.2.1. What Are Bash Tests?

Tests in bash are constructs that allow you to do conditional expressions. They use the square brackets to enclose what is being tested.

For example, the simplest tests might be:

```
1$ [ false = true ]
2$ echo $?
3$ [ true = true ]
4$ echo $?
```

> The 'echo $?' command above is a little mystifying at this stage if you've not seen it before. We will cover it in more depth in a section later in this part. For now, all you need to understand is this: the '$?' variable is a special variable that gives you a number telling you whether the last-executed command succeeded or not. If it succeeded, the number will (usually) be '0'. If it failed, the number will (usually) *not* be '0'.

'false' is a program that has one job: to produce a 'failing' exit code. Why would such a thing exist? Well, it's often useful to have a command that you know will fail when you are building or testing a script.

Things get more interesting if you try and compare values in your tests. Think about what this will output before typing it in:

```
5$ A=1
6$ [ $A = 1 ]
7$ echo $?
8$ [ $A == 1 ]
9$ echo $?
10$ [ $A = 2 ]
11$ echo $?
```

A single equals sign works just the same as a double equals sign. Generally I prefer the double one so it does not get confused with variable assignment.

2.2.2. What is '[', Really?

It is worth noting that '[' is in fact a builtin, as well as (very often) a program.

```
12$ which [
13$ builtin [
```

i. and that '[' and 'test' are synonymous

```
14$ which test
15$ builtin test
```

'which' is a program (not a builtin!) that tells you where a program can be found on the system.

This is why a space is required after the '['. It's a separate command and spacing is how bash determines where one command ends and another begins.

2.2.3. Logic operators

What do you expect the output of this to be?

```
16$ ( [ 1 = 1 ] || [ ! '0' = '0' ] ) && [ '2' = '2' ]
17$ echo $?
```

Similar to other languages, '!' means 'not', '||' means 'or', '&&' means 'and' and items within '()' are evaluated first.

Note that to combine the binary operators '||' and '&&' you need to have separate '[' and ']' pairs.

If you want to do everything in *one* set of braces, you can run:

```
18$ [  1 = 1 -o  ! '0' = '0'  -a  '2' = '2' ]
19$ echo $?
```

You can use '-o' as an 'or' operator within the square brackets, -a for 'and' and so on. But you can't use '(' grouping within them.

If you're not confused yet, you might be soon!

2.2.4. [[

There is another very similar operator to the 'test' one that has *two* angle brackets:

```
20$ [[ 1 = 1 ]]
21$ echo $?
```

This confused me a lot for some time! What's the difference between then?

The differences between '[[' and '[' are relatively subtle. Type these lines to see examples:

```
22$ unset DOESNOTEXIST
23$ [ ${DOESNOTEXIST} = '' ]
24$ echo $?
25$ [[ ${DOESNOTEXIST} = '' ]]
26$ echo $?
27$ [ x${DOESNOTEXIST} = x ]
28$ echo $?
```

The first command (22) errors because the variable 'DOESNOTEXIST'... does not exist. So bash processes that variable and ends up running:

```
[ = '' ]
```

which makes no sense, so it complains!

The second command (24), which uses the double brackets, tolerates the fact that the variable does not exist, and treats it as the empty string.

The third command acts as a workaround, by placing an 'x' on both sides of the equation to ensure that *something* gets

You frequently come across code like this:

```
29$ [[ "x$DOESNOTEXIST" = "x" ]]
```

where users have put quotes on both sides as well as an x and put in double brackets. Only one of these protections is needed, but people get used to adding them on as superstitions to their bash scripts.

Once again, you can see understanding how quotes work is critical to bash mastery!

Oh, and '[[' doesn't like the '-a' (and) and '-o' (or) operators.

So '[[' can handle some edge cases when using '['. There are some other differences, but I won't cover them here.

> If you want to understand more, follow [this link](http://serverfault.com/questions/52034/what-is-the-difference-between-double-and-single-square-brackets-in-bash)

2.2.5. Confused?

You're not alone. In practice, I follow most style guides and always use '[[' until there is a good reason not to.

If I come across some tricky logic in code I need to understand, I just look it up there and then.

2.2.6. Unary and Binary Operators

There are other shortcuts to test that it's worth knowing about. These take a single argument:

```
30$ echo $PWD
31$ [ -z "$PWD" ]
32$ echo $?
33$ unset DOESNOTEXIST
34$ [ -z "$DOESNOTEXIST" ]
35$ echo $?
36$ [ -z ]
37$ echo $?
```

If your $PWD environment variable is set (it usually is), then the -z will return 'false'. This is because '-z' returns true only if the argument is an empty string. Interestingly, this test is OK with no argument! Just another confusing point about tests...

There are quite a few unary operators so I won't cover them all here. The ones I use most often are '-a' and '-d':

```
38$ mkdir lbthw_tmp_dir
39$ touch lbthw_tmp_file
40$ [ -a lbthw_tmp_file ]
41$ echo $?
42$ [ -d lbthw_tmp_file ]
43$ echo $?
44$ [ -a lbthw_tmp_dir ]
45$ echo $?
46$ [ -d lbthw_tmp_dir ]
47$ echo $?
48$ rm lbthw_tmp_dir lbthw_tmp_file
```

These are called 'unary operators' (because they take one argument).

There are many of these unary operators, but the differences between them are useful only in the rare cases when you need them. Generally I just use '-d', '-a', and '-z' and look up the others when I need something else.

We'll cover 'binary operators', which work on two arguments, while covering types in bash.

2.2.7. Types

Type-safety (if you're familiar with that from other languages) does not come up often in bash as an issue. But it is still significant. Try and work out what's going on here:

```
49$ [ 10 < 2 ]
50$ echo $?
51$ [ '10' < '2' ]
52$ echo $?
53$ [[ 10 < 2 ]]
54$ echo $?
55$ [[ '10' < '2' ]]
56$ echo $?
```

From this you should be able to work out that the '<' operator expects strings, and that this is another way '[[' protects you from the dangers of using '['.

If you can't work it out, then re-run the above and play with it until it makes sense to you!

Then run this

```
57$ [ 10 -lt 2 ]
58$ echo $?
59$ [ 1 -lt 2 ]
60$ echo $?
61$ [ 10 -gt 1 ]
62$ echo $?
63$ [ 1 -eq 1 ]
64$ echo $?
65$ [ 1 -ne 1 ]
66$ echo $?
```

The binary operators used above are: '-lt' (less than), '-gt' (greater than), '-eq' (equals), and '-ne' (not equals). They deal happily with integers in single bracket tests.

2.2.8. if statements

Now you understand tests, if statements will be easy:

```
67$ if [[ 10 -lt 2 ]]
then
  echo 'does not compute'
elif [[ 10 -gt 2 ]]
then
  echo 'computes'
else
  echo 'does not compute'
fi
```

'if' statements consist of a test, followed by the word 'then', the commands to run if that if returned 'true'. If it returned false, it will drop to the next 'elif' statement if there is another test, or 'else' if there are no more tests. Finally, the if block is closed with the 'fi' string.

The 'else' or 'elif' blocks are not required. For example, this will also work:

```
68$ if [[ 10 -lt 2 ]]; then echo 'does not compute'; fi
```

as the newline can be replaced by a semi-colon, which indicates the end of the expression.

2.2.9. What You Learned

We covered quite a lot in this section!

- What a 'test' is in bash
- How to compare values within a test
- What the program '[' is
- How to perform logic operations with tests
- Some differences between '[' and '[['
- The difference between unary and binary operators
- How types can matter in bash, and how to compare them
- 'if' statements and tests

2.2.10. What Next?

Next you will cover another fundamental aspect of bash programming: loops.

2.2.11. Exercises

1) Research all the unary operators, and try using them (see 'man bash')

2) TODO

2.3. Loops

Like almost any programming language, bash has loops.

In this section you will cover for loops, case statements, and while loops in bash.

This section will quickly take you through the various forms of looping that you might come across.

2.3.1. 'for' Loops

First you're going to run a for loop in a 'traditional way':

```
1$ for (( i=0; i < 20; i++ ))
do
  echo $i
  echo $i > file${i}.txt
done
2$ ls
```

You just created twenty files, each with a number in them using a 'for' loop in the 'C' language style. Note there's no '$' sign involved in the variable when it's in the double parentheses!

```
3$ for f in $(ls *txt)
do
  echo "File $f contains: $(cat $f)"
done
```

It's our old friend the subshell! The subshell lists all the files we have.

This for loop uses the 'in' keyword to separate the variable each iteration will assign to ('f') and the list to take items from. Here bash evaluates the output of the 'ls' command and uses that as the list, but we could have written something like:

```
4$ for f in file1.txt file2.txt file3.txt
do
  echo "File $f contains: $(cat $f)"
done
```

with a similar effect.

2.3.2. 'while' and 'until'

While loops also exist in bash. Try and work out what's going on in this trivial example:

```
5$ n=0
6$ while [[ ! -a newfile ]]
do
    echo "In iteration $n"
    if [[ $(cat file${n}.txt) == 15 ]]
    then
        touch newfile
    fi
done
7$ echo "done"
```

I often use while loops in this 'infinite loop' form when running quick scripts on the command line:

```
8$ n=0
9$ while true
do
    echo $n seconds have passed
    sleep 1
    if [[ $n -eq 60 ]]
    then
        break
    fi
done
```

2.3.3. 'case' Statements

Case statements may also be familiar from other languages. In bash, they're most frequently used when processing command-line arguments within a script.

Before you look at a realistic case statement, type in this trivial one:

```
10$ a=1
11$ case "$a" in
1) echo 'a is 1'; echo 'ok';;
2) echo 'a is 2'; echo 'ok';;
*) echo 'a is unmatched'; echo 'failure';;
esac
```

Try triggering the 'a is 2' case, and the 'a is unmatched' case.

There are a few of new bits of syntax you may not have seen before.

First, the double semi-colons ';;' indicate that the next matching case is coming (rather than just another statement, as indicated by a single semi-colon).

Next, the '1)' indicates what the 'case' value ("$a") should match. These values follow the globbing rules (so '*' will match anything). Try adding quotes around the values, or glob values, or matching a longer string with spaces.

Finally, the 'esac' indicates the case statement is finished.

2.3.4. 'case' Statements and Command Line Options

Case statements are most often seen in the context of processing command-line options within shell scripts. There is a helper builtin just for this purpose: getopts.

Now you will write a more realistic example, and more like what is seen in 'real' shell scripts that uses getopts. Create and move into the folder:

```
11$ mkdir lbthw_loops
12$ cd lbthw_loops
```

Now create a file (case.sh) to try out a case statement with getopts:

```
13$ cat > case.sh << 'EOF'
#!/bin/bash
while getopts "ab:c" opt
do
    case "$opt" in
    a) echo '-a invoked';;
    b) echo "-b invoked with argument: ${OPTARG}";;
    c) echo '-c invoked';;
    esac
done
EOF
14$ chmod +x case.sh
```

Run the above with various combinations and try and understand what's happening:

```
15$ ./case.sh -a
16$ ./case.sh -b
17$ ./case.sh -b "an argument"
18$ ./case.sh -a -b -c
19$ ./case.sh
```

This is how many bash scripts pick up arguments from the command line and process them.

2.3.5. Cleanup

```
20$ cd ..
21$ rm -rf lbthw_loops
```

2.3.6. What You Learned

You've now covered the main methods of looping in bash. Nothing about looping in bash should come as a big surprise in future!

2.3.7. What Next?

Next you will learn about exit codes, which will power up your ability to write neater bash code and better scripts.

2.3.8. Exercises

1) Find a real program that uses getopts to process arguments and figure out what it's doing.

2) TODO

2.4. Exit Codes

Now that you know about tests and special parameters in bash, a crucial and related thing to understand is exit codes.

2.4.1. What Is An Exit Code?

When you run a command, function or builtin, a special variable is set that tells you what the result of that command was. If you're familiar with HTTP codes like 200 or 404, it's a similar concept to that.

Take a simple example:

```
1$ ls
2$ echo $?
3$ doesnotexist
4$ echo $?
```

When that special variable is set to '0' it means that the command completed successfully.

You should be able to follow what is going on here at this point:

```
5$ bash
6$ function trycmd {
    $1
    if [[ $? -eq 127 ]]
    then
        echo 'What are you doing?'
    fi
}
7$ trycomd ls
8$ trycmd doesnotexist
9$ exit
```

You can easily write tests to use exit codes for various ends like this.

2.4.2. Standard Exit Codes

There are guidelines for exit codes for those that want to follow standards.

Some key ones are:

Number	Meaning	Notes
0	OK	Command successfully run
1	General error	Used when there is an error but no specific number reserved to indicate what it was
2	Misuse of shell builtin	
126	Cannot execute	Permission problem or command is not executable
127	Command not found	No file found matching the command
128	Invalid exit value	Exit argument given (eg 'exit 1.76')
128+n	Signal 'n'	Process killed with signal 'n', eg 130 = terminated with CTRL-c (signal 2)
255	Exit code out of range	

Signals will be covered in part 4

Since codes 3-125 are not generally reserved, you might use them for your own purposes in your application.

2.4.3. Exit Codes and if Statements

So far so simple, but unfortunately (and because they are useful) exit codes can be used for many different reasons, not just to tell you whether the command completed successfully or not. Just as with exit codes in HTTP, the application can use exit codes to indicate something went wrong, or it can return a '200 OK' and give you a message directly.

Try to predict the output of this:

```
10$ echo 'grepme' > afile.txt
11$ grep not_there afile.txt
12$ echo $?
```

Did you expect that? grep finished successfully (there was no segmentation fault, memory, it was not killed etc..) but no lines were matched to it returned '1' as an exit code.

In one way this is great because you can write if statements like this:

```
13$ if grep grepme afile.txt
then
    echo 'matched!'
fi
```

On the other hand, it means that you cannot be sure about what an exit code might mean about a particular program's termination. I have to look up the grep exit code every time, and if I use a program's exit code I make sure to do a few tests first to be sure I know what is going to happen!

2.4.4. Setting Your Own Exit Code

If you are writing a script or a function, you can set the exit code by using the 'return' builtin.

Type this:

```
14$ bash
15$ function trycmd {
    $1
    if [[ $? -eq 127 ]]
    then
        echo 'What are you doing?'
        return 1
    fi
}
16$ trycmd ls
17$ trycmd doesnotexit
18$ exit
```

2.4.5. Other Special Parameters

The variable '$?' is an example of a 'special parameter'. I'm not sure why they are called 'special parameters' and not 'special variables', but it is perhaps to do with the fact that they are considered alongside the normal parameters of functions and scripts ($1,$2 etc) as automatically assigned variables within a context.

Two of the most important are used in the below listing. Figure out what they are:

```
19$ ps -ef | grep bash | grep $$
20$ sleep 999 &
21$ echo $!
```

If you're still stuck, have a look at the bash man page...

Cleanup

Now clean up what you just did:

```
22$ rm afile.txt
```

2.4.6. What You Learned

- What an exit code is
- Some standard exit codes and their significance
- How tests and exit codes work together
- Some special parameters

2.4.7. What Next?

Next you will learn about bash options, and the 'set' builtin.

2.4.8. Exercises

1) Look up under what circumstances git returns a non-zero exit code.

2) Look up all the 'special parameters' and see what they do. Play with them. Research under what circumstances you might want to use them.

2.5. The 'set' Builtin

When using bash it is very important to understand what options are set, and how this can affect the running of your scripts.

In this section you will become familiar with the 'set' builtin, which allows you to manipulate these options within your scripts.

2.5.1. Running 'set'

You start by running 'set' on its own:

```
1$ set
```

This will produce a stream of output that represents the state of your shell. In the normal case, you will see all the variables and functions set in your environment.

But my bash man page says:

> Without options, the name and value of each shell variable are displayed in a format that can be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In posix mode, only shell variables are listed.
>
> — bash man page

The **Portable Operating System Interface(POSIX)** is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Can you work out from your 'set' output whether you are in posix mode?

It is likely that you are not. If so, type:

```
2$ bash
3$ set -o posix
4$ set
5$ exit
```

and you will observe that the output no longer has functions in it. The '-o' switched on the 'posix' option in your bash shell. The same command with '+o' will switch it off. I have trouble remembering which is 'on' and which is 'off' every time!

If you did not have functions, then either no functions were set, or you were in posix mode already!

The commands above put you in a fresh bash shell so that we would revert to the previous state.

To show how all your options are set type this:

```
6$ set -o
```

and you will see the current state of all your options.

What you see are all the options bash can set. One of the exercises below is to try to understand what they all mean, but in this section we're only going to focus on a couple that I use all the time.

2.5.2. 'set' vs 'env'

One thing that can confuse people is that the output of 'set' is similar to the output of 'env', but different.

```
7$ set
8$ env
```

The difference is that *exported* variables are shown by env, not all the variables set in the shell.

2.5.3. Useful Options for Scripting

Where set becomes really useful to understand is in scripting.

For example, I set these two up every time I start writing a shell script:

```
9$ set -o errexit
10$ set -o xtrace
```

Although you don't need to be in a script for them to work.

The errexit option tells bash to exit the script if any command fails.

The xtrace option outputs the

Type this is to see how this works in practice:

```
11$ echo '#!/bin/bash
set -o errexit
set -o xtrace
pwd
cd $HOME
cd -
false
echo "should not get here"' > ascript.sh
12$ chmod +x ascript.sh
13$ ./ascript.sh
```

You should be able to explain the output of the above.

Flags With 'set' Instead of Names

For each 'set' option, you can use a flag instead. For example, this:

```
14$ set -e
15$ set -x
```

is the same as:

```
16$ set -o errexit
17$ set -o xtrace
```

I generally prefer the name form rather than flag, just because it's easier to read.

When I start writing a script, I usually start with the following:

```
#!/bin/bash

set -o errexit
set -o xtrace
set -o unset

[...]
```

The 'unset' throws an error if a variable is unset when referenced. (The special variables $@ and $* are exempt).

2.5.4. The 'pipefail' Option, Exit Codes and Pipelines

One option worth mentioning (as it is frequently referred to) is the pipefail option:

```
18$ set -o pipefail
18$ grep notthere afile.txt | xargs echo
19$ echo $?
20$ set +o pipefail
21$ grep notthere afile.txt | xargs echo
22$ echo $?
```

One of the above yields a '0' exit code, and the other does not. The only difference is the pipefail setting.

When switched on (remember, '-o' is on, '+o' is off - yes, I find it confusing too), the pipefail returns the error code of the last command to return a non-zero status. Since grep returns non-zero even when there's no 'error' as such, you can get surprising behaviour when using pipes and exit codes.

By default, pipefail is off, so the second outcome is the default one.

Cleanup

Now clean up what you just did:

```
23$ rm afile.txt
24$ rm ascript.sh
```

2.5.5. What You Learned

- What the 'set' builtin is
- How to set an option
- The difference between -o (on) and +o (off)
- Some of the most-used and useful options

2.5.6. What Next?

Well done! You've made it to the end of the scripting section. Now you are fully equipped to write and read useful shell scripts.

The next part looks at the most-frequently used idioms and tricks commonly-seen in bash scripts.

2.5.7. Exercises

1) Read the man page to see what all the options are. Don't worry if you don't understand it all yet, just get a feel for what's there.

2) Set up a shell with unique variables and functions and use set to create a script to recreate those items in another shell.

Chapter 3. Tips

In this section I cover concepts and techniques that form the basis of daily usage of bash. They can be considered 'tips' for bash usage and somewhat optional, but come up often enough for me to think it important to cover in a practical guide.

In it we cover:

- Terminal codes and non-standard characters
- Setting up your prompt
- 'Here' docs
- The bash command history

3.1. Terminal Codes and Non-Standard Characters

Although not directly related to bash, if you spend any time at a terminal, then it will pay off to understand how the terminal works with non-standard characters.

Non-standard characters are characters like 'tab', 'newline', 'carriage return', and even 'end of file'. They don't form part of words, or necessarily print anything to the screen, but they are interpreted by the shell and the terminal.

In this section you'll:

- Understand how to output anything with echo
- Understand how to output a character while by-passing the terminal
- Learn about carriage returns and newlines and how they are used
- Learn how to use hexdump
- Learn about standard terminal escape codes

> Focus here is on ANSI-standard escapes. Rarely, you might come across more complex escapes for specific terminal contexts, but this is beyond the scope of a 'practical' guide.

3.1.1. Non-Printable Characters

The terminal you use has what are described as 'printable' characters, and non-printable characters.

For example, typing a character like 'a' (normally) results in the terminal adding an 'a' to the screen. But there are other characters that tell the terminal to do different things.

It's easy to forget this, but not everything the shell receives is directly printed to the screen. The terminal 'driver' takes what it is given (which is one or more bytes) and decides what to do with it. It might decide to print it (if it's a 'normal' character), or it might tell the computer to emit a beep, or it might change the colour of the terminal, or delete and go back a space, or it might send a message

to the running program to tell it to exit.

When looking at odd characters, it's useful to be aware of a couple of utilities that help you understand what's going on. The first of these is a familiar one: echo.

3.1.2. Using 'echo'

You're already familiar with echo, but it has some sometimes-useful flags:

```
1$ echo -n 'JUST THESE CHARACTERS'
```

The -n flag tells echo not to print a newline

```
2$ echo -n -e 'JUST THESE CHARACTERS AND A NEWLINE\n'
```

The '-e' makes us sure that the '\n' will create a newline. Most often it's the default anyway.

It is the backslash character '\' that makes echo aware that a special character is coming up.

Other special characters include '\b' (for backspace), '\t' (for tab) and '\\' (to output a backslash):

What do you think this will output?

```
3$ echo -n -e 'a\bb\bcde\bfg\b\b\b\n'
```

You can also send a specific byte value to the terminal by specifying its hex value:

```
4$ echo -n -e '\x20\n'
```

Think about that - you can use echo to control *exactly* what gets sent to the screen. This is extremely valuable for debugging.

If you want to *prevent* this behaviour, use the '-E' flag:

```
5$ echo -E -n 'Line1\nLine2'
```

3.1.3. CTRL-v Escaping

Being able to output any binary value to the screen that we choose is useful, but what if we want to just output a 'special' character, and not have the terminal interpret it in any special way?

For example, if I hit 'tab' in my terminal it would normally *not* show a tab character, as the terminal uses the tab key (if hit twice quickly) to auto-complete any text we have not finished (we will cover this later). Try it!

But if I'm typing something like:

```
6$ echo -E 'I want a tab here:>X<a tab'
```

How do I get a tab where the 'X' is?

You do it like this: instead of the 'X', you hold down the 'CTRL' key. Then hold down the 'v' key. Let go of the 'v' key, then tap the 'i' key. Finally, let go of the 'CTRL' key.

Practice that a few times. The pattern is: hit CTRL+v, then CTRL+the character specified (in this case 'i').

This character would normally be represented like this: '^I', the '^' indicating a 'CTRL+V' combination, and the 'I' indicating the CTRL+I combination.

How would you input this, therefore, and what will it output?

```
7$ echo abcc^Hdefg
```

This table is a useful reference for these characters:

Name	Hex	C-escape	CTRL-key	Description	
BEL	0x07	\a	^G	Terminal bell	
BS	0x08	\b	^H	Backspace	
HT	0x09	\t	^I	Horizontal TAB	
LF	0x0A	\n	^J	Linefeed	
VT	0x0B	\v	^K	Vertical TAB	
FF	0x0C	\f	^L	Formfeed	
CR	0x0D	\r	^M	Carriage return	
ESC	0x0E	N/A	^[	Escape character	
DEL	0x0F	N/A	N/A	Delete character	

There are other interesting CTRL-v escape characters, but they are more rarely used. We won't cover them here.

3.1.4. Carriage return vs Line Feeds

The most commonly-seen

Carriage returns and line feeds cause much confusion, but it doesn't take long to understand the difference and why they are different.

If you think about an old-fashioned typewrite or printer that moves along punching out characters to a page, at some point it has to be told: 'go back to the beginning of the line'. Then, once at the beginning of the line, it has to be told: 'feed the paper up one line so I can start writing my new line'.

A 'carriage return' is, as the word 'return' suggests, 'returns' the cursor to the start of the line. It's represented by the character 'r' for return. The 'line feed', again as the name suggests, feeds the line up. In a modern terminal, this just means 'move the cursor down'.

So far, so clear and simple to learn. But, Linux does things differently! In Linux, '\n' is sufficient to do both. In Windows, you need both characters to represent a new line.

What will this output?

This means that files can 'look funny' in Linux terminals with these weird '^M' characters showing at the end of each line. To confuse things even more, some programs automatically handle the difference for you and hide it from you.

```
8$ echo -e 'Bad magazine\rMad'
```

This is why it's important to have a way to see what the actual bytes in a file are, and where a very useful tool comes in: hexdump.

3.1.5. Hexdump

Run this:

```
9$ echo -e 'Bad magazine\rMad' | hexdump
10$ echo -e 'Bad magazine\rMad' | hexdump -c
```

Hexdump prints out the characters received in standard input as hex digits. 16 characters are printed per line, and on the left is displayed the count (also in hex) of the number of bytes processed up to that line.

The -c flag prints out the contents as characters (including the control ones with appropriate backslashes in front, eg '\n', whereas leaving it out just displays the hex values.

It's a great way to see what is *really* going on with text or any stream of output of bytes.

If you go back to the first example in this section:

```
11$ echo 'JUST THESE CHARACTERS' | hexdump -c
12$ echo -n 'JUST THESE CHARACTERS' | hexdump -c
```

You can figure out for yourself the difference between using the -n flag in echo and not using it.

3.1.6. Terminal Escape Codes

Run this:

```
13$ echo -e '\033[?47h'
14$ echo -e '\033[?47l'
```

The first line 'saves' the screen (but does not clear it!) and the second restores it.

These terminal escape codes are standard sequences that tell the terminal to do various things.

The ANSI codes always start with the ESC character and left bracket character: in hex '1B' then '5b', or in octal '033' then ''. So you could rewrite the above as:

```
15$ echo -e '\x1b\x5b?47h'
16$ echo -e '\x1b\133?47l'
```

These characters are then followed by specific sequences which can change the colour of the screen, the background text, the text itself, set the screen width, or even re-map keyboard keys.

Type this out and see if you can figure out what it's doing as you go:

```
17$ ansi-test() {
for a in 0 1 4 5 7
    do
    echo "a=$a "
    for (( f=0; f<=9; f++ ))
    do
        for (( b=0; b<=9; b++ ))
        do
            echo -ne "\\033[${a};3${f};4${b}m"
            echo -ne "\\\\\\\\033[${a};3${f};4${b}m"
            echo -ne "\\033[0m "
        done
        echo
    done
    echo
done
echo
}
```

That shows you what all the ansi terminal escape codes are and you can see what they do in the terminal.

Sometimes when you 'cat' a binary file, (or /dev/random, which outputs random bytes) the contents when output to a terminal can cause the terminal to appear to 'go haywire'. This is because these escape codes are accidentally triggered by the sequences of bytes that happen to exist in these files.

3.1.7. Fun With Terminals

Finally, some (optional) fun which pulls together a few different things you've learned along the way.

Create this as a file called 'shiner', and run it with:

```
sh shiner
```

and remove it afterwards if you like.

```
#!/bin/bash

DATA[0]=" _/ _/ _/ _/ "
DATA[1]=" _/_/_/_/_/ _/_/_/ _/_/_/ _/_/_/ _/_/_/ "
DATA[2]=" _/ _/ _/ _/ _/ _/ _/_/ _/ _/"
DATA[3]="_/_/_/_/_/ _/ _/ _/ _/ _/_/ _/ _/ "
DATA[4]=" _/ _/ _/_/_/ _/_/_/ _/_/_/ _/ _/ "

REAL_OFFSET_X=0
REAL_OFFSET_Y=0

draw_char() {
   V_COORD_X=$1
   V_COORD_Y=$2

   tput cup $((REAL_OFFSET_Y + V_COORD_Y)) $((REAL_OFFSET_X + V_COORD_X))

   printf %c ${DATA[V_COORD_Y]:V_COORD_X:1}
}

trap 'exit 1' INT TERM
trap 'tput setaf 9; tput cvvis; clear' EXIT

tput civis
clear

while :; do
    for ((c=1; c <= 7; c++)); do
        tput setaf $c
        for ((x=0; x<${#DATA[0]}; x++)); do
        for ((y=0; y<=4; y++)); do
        draw_char $x $y
        done
        done
    done
done
```

3.1.8. What You Learned

- What terminal codes are
- What printable and non-printable characters are
- How to output any arbitrary item
- How to prevent the terminal from interpreting the character using CTRL-v
- The difference between '\n' and '\r\n'
- What terminal escape codes are

3.1.9. What Next?

Building on this knowledge, next you will learn how to set up your prompt so that it can show you (and even do) useful things.

3.1.10. Cleanup

You don't necessarily need to clean up at the end of this section, but your terminal may have inadvertently changed state if input was wrongly made.

If this happens, kill or exit your terminal and restart bash.

3.1.11. Exercises

1) Research and echo all of echo's escape sequences. Play with them and figure out what they do.

2) Research and echo 10 terminal escape sequences.

3) Look up all the CTRL-v escape sequences and experiment with them.

4) Research the command 'tput', figure out what it does and rewrite some of the above commands using it.

5) Re-map your keyboard so it outputs the wrong characters using escape codes.

3.2. The Prompt

Now that you've learned about escapes and special characters you are in a position to understand how the prompt can be set up and controlled.

3.2.1. The PS1 Variable

Type this:

```
1$ bash
2$ PS1='My super prompt>>>>   '
3$ ls
4$ exit
```

As you'll remember, there are a bunch of

3.2.2. The PS2 Variable

Now try this:

```
5$ bash
6$ PS2='I am in the middle of something!>>>  '
7$ cat > /dev/null << END
some text
END
8$ exit
```

The PS2 variable is the 'other' prompt, that the shell uses to indicate that you are being prompted for input to a program that is running.

3.2.3. PS3 and PS4

PS3 is used by the 'select' looping structure. We don't cover that in this section.

PS4 is the last one:

```
9$ bash
10$ PS4='> Value of PWD is: $PWD'
11$ set -x
12$ pwd
13$ cd /tmp
14$ ls $(pwd)
15$ cd -
16$ exit
```

In 'trace' mode PS4 is echoed before each line of trace output.

But why is the '>' in echo repeated? This indicates the level of indirection (eg subshells) in the trace. Every time the script goes one level 'down', the first character in the PS3 value is repeated. Look at the output after the 'ls $(pwd)' command.

> Things can get really confusing if you have commands in your prompt, or you have PROMPT_COMMAND set (see below section). If you don't fully understand the output of the above, don't panic!

3.2.4. Pimp Your Prompt

For all the PS variables mentioned above, there are special escape values that can be used to make your prompt display interesting information.

See if you can figure out what is going on here:

```
17$ bash
18$ PS1='\u@\H:\w \# \$ '
19$ ls
20$ exit
```

The table below may help you.

Escape value	Meaning	Notes
\#	Command number	The number (starting from 1 and incrementing by one) of the command in this bash session.
\$	Root status	If you have root, show a '#' sign, otherwise show '$'
\t	Current time	In HH:MM:SS format - there are other formats possible with eg \T.
\H	Hostname	The hostname (fully-qualified)
\w	Current working directory	
\[	Start control sequence	Begin a sequence of non-printing characters, eg put a terminal control sequence in a prompt.
\]	End control sequence	

Use your knowledge gained so far to figure out what is going on here:

```
21$ bash
22$ PS1='\[\033[01;31m\]PRODUCTION\$ '
23$ PS1='\[\033[01;32m\]DEV\$ '
24$ exit
```

How would you make this automatically happen on a given server when you log in?

3.2.5. PROMPT_COMMAND

Another way the prompt can be affected is with the bash variable 'PROMPT_COMMAND':

```
25$ bash
26$ PROMPT_COMMAND='echo "Hello prompt $(date)"'
27$ ls
28$ exit
```

Every time the prompt is displayed the PROMPT_COMMAND is treated as a command, and run.

You can use this for all sorts of neat tricks!

3.2.6. What You Learned

- What the PS variables are
- Where each PS variable is used one
- How to augment your prompts to give you useful information
- How to automatically run commands before each prompt is shown

3.2.7. What Next?

Next you will learn a very useful technique for quickly creating files: the so-called 'here doc'.

3.2.8. Cleanup

No cleanup is required here, though you may want to set up a fresh bash session in case your prompt has been changed.

3.2.9. Exercises

1) Look up the other prompt escape characters and use them.

2) Update your bash startup files so that the prompt tells you useful information when you log in

3) Create your own version of the 'history' command by using the 'PROMPT_COMMAND' variable.

3.3. Here Documents

Here documents are one of the handiest tricks you will have at your disposal. They allow you to embed content directly in your script.

> In this section, leading spaces are tabs. We have covered this in a previous section, but as a reminder: to get a tab character in your shell, type 'CTRL+v', and then hit the 'tab' button.

3.3.1. Basic Here Docs

Type this in to see the basic form of the here doc:

```
1$ cat > afile.txt << END
A file can contain
    whatever you like
END
2$ cat afile.txt
```

The 'cat' command is used without a filename given, and the output is redirected to the file 'afile.txt'. If no filename is given, cat takes its input from standard input.

The '<<' indicates that the standard input will be taken up to whatever line just contains the word after the '<<' characters. In this case, the word is 'END'.

The word does not need to be 'END'! It could be anything you choose. 'END' is generally used as a convention. Sometimes you see 'EOF', or 'STOP', or something similar. If you have a document with 'END' in it, for example, you might want to avoid problems with the document ending early by choosing a different word.

3.3.2. More Advanced Here Docs

Now you're going to put a here document in a function. The function takes one argument. This argument is used as a filename, and the function creates a simple script with that filename that echoes the first argument given to that script.

Will this work? Read it carefully, predict the outcome, and then run it:

```
3$ function write_echoer_file {
    cat > $1 << END
#!/bin/bash
echo $1
END
    chmod +x $1
}
4$ write_echoer_file echoer
5$ ./echoer 'Hello world'
```

Hmmm. That didn't work, because the '$1' got interpreted in the write_echoer_file function as being the filename we passed in. In the 'here doc', we wanted the '$1' characters to be put into the script without being interpreted.

Try this instead:

```
6$ function write_echoer_file {
    cat > $1 << 'END'
#!/bin/bash
echo $1
END
    chmod +x $1
}
7$ write_echoer_file echoer
8$ ./echoer 'Hello world'
```

Do you see the difference? This time, the delimiter word 'END' was wrapped in single quotes. This made sure that the 'echo $1' was not interpreted by the shell when being written in

Can you see why we needed to use single quotes here? What happens when you use double quotes?

This kind of confusion can happen all the time when writing bash scripts, so it's really important to get these differences clear in your mind.

Our function is working now, but we could still make it better.

Try this (remember, the leading spaces are tabs - see the note above for how to input a tab):

```
9$ function write_echoer_file {
    cat > $1 <<- 'END'
        #!/bin/bash
        echo $1
    END
    chmod +x $1
}
10$ write_echoer_file echoer
11$ ./echoer
```

What if END is part of the here doc?

```
12$ function write_echoer_file {
    cat > $1 <<- 'END'
        #!/bin/bash
        echo $1
        echo Is this the END?
    END
    chmod +x $1
}
13$ write_echoer_file echoer
14$ ./echoer
```

No problem if it is not the only thing on the line:

3.3.3. Here Strings

Related to the 'here doc', a 'here string' can be applied in the same way with the '<<<' operator:

```
15$ function write_here_string_to_file {
    cat > $1 <<< $2
}
16$ write_here_string_to_file afile.txt "Write this out"
```

3.3.4. Cleanup

```
17$ rm -f echoer afile.txt
```

3.3.5. What You Learned

- What 'here documents' are
- What 'here strings' are
- How to create a 'here document'
- How here docs and variables can be appropriately handled
- How to use here docs in a way that looks neat in a shell script

3.3.6. What Next?

Next we look at how bash maintains and uses a history of the commands run within it.

3.3.7. Exercises

1) Try passing a multi-line string to a here string. What happens?

Chapter 4. History

We all know understanding history is important, and this is true in bash as well.

This section gives you a pragmatic overview of bash's history features, which can save you lots of time when at the terminal.

4.1. Bash and History

Bash keeps a history of commands you have run in a file. By default this is in your $HOME directory and has the name '.bash_history'.

Have a look at it:

```
1$ cat ~/.bash_history
```

4.2. Using Your History

It can be tedious to type out commands and arguments again and again, so bash offers several ways to save your effort.

Type this out and try and figure out what is going on:

```
2$ mkdir lbthw_history
3$ cd !$
4$ echo 'About bash history' > file1
5$ echo 'Another file' > file2
6$ grep About file1
7$ !!
8$ grep About file2
9$ grep Another !$
10$ rm file2
11$ !e
12$ !gr
```

That introduced a few tricks you haven't necessarily seen before.

All of them start with the '!' (or so-called 'bang') sign, which is the sign used to indicate that the bash history is being referred to.

The simplest, and most frequently seen is the double bang '!!', which just means: re-run the previous command.

The one I use most often, though, is the first one you come across in the listing above: '!$', or 'bang dollar'. This one I must use dozens of times every day. It tells bash to re-use the last argument of the previous command.

Finally, a 'bang' followed by 'normal' characters re-reruns the last command that matches those starting letters. The '!e' looks up the last command that ran starting with an 'e' and runs that. Similarly, the '!gr' runs the last command that started with a 'gr', ie the grep.

Notice that the command that's rerun is the *evaluated* command. For that grep, what is re-run is as though you typed: 'grep Another file2', and not 'grep Another !$'.

4.3. How to Learn Them

The history items above are enough to be going on with if you've not seen them before.

Before you go on, a quick note about learning these things: it's far more important to learn to *use* these tricks than *understand* them. To understand them is pretty easy - I'm sure you understood the passage above without much difficulty.

The way to learn these is to 'get them under your fingers' to the point where you don't even think about it. The way I recommend to do that is to concentrate on one of them at a time, and as you're working, remember to use that one where appropriate. Gradually you'll add more and more to your repertoire, and you will soon look like a whizz at the terminal.

4.3.1. More Advanced History Usage

You might want to stop there, as trying to memorize/learn much more in one go can be overwhelming.

But there are many more tricks to learn like this in bash, so I'm going to lay them out now so you might return to them later when you're ready.

Carrying on from where you left off above:

```
13$ grep Abnother file1
14$ ^Ab^a^
```

The caret ('^') are used to replace a string from the previous command. In this case, 'Ab' is replaced with 'a'. This is often handy if you made a spelling mistake.

Next up are the position command shortcuts, or 'word designators':

```
15$ grep another file1 | wc -l
16$ # Is that output correct? I want to check the file by eye:
17$ cat !:2
```

Starting with the 'bang' sign to indicate we're referring to the history, there follows a colon. Then, you specify the word with a number. The numbers are zero-based, so

```
18$ grep another file1
19$ fgrep !:1-$
```

In the above example, you want to run the same command as before, but use the 'fgrep' command instead of 'grep' ('fgrep' is a 'faster' grep, which doesn't really help us here, but is just an example). To achieve this you use the so-called 'word designators'.

Here you add a dash indicates you want a range of words, and the '$' sign indicates we want all the arguments up to the end of the previous command. Recall that '!$' means give me the last argument from the previous command, and so is itself a shortcut for '!:$'.

Finally, another trick I use all the time:

```
21$ LGTHWDIR=$(PWD)
22$ cd /tmp
23$ cat ${LGTHWDIR}/file1
24$ cd !$:h
```

The trick is the ':h' modifier. This is one of several modifiers available, but the only one I regularly use. When using a history shortcut, you can place a modifier at the end that starts with a colon. Here, the '!$' takes the last word from the previous command, (which you set to full directory path to the freshly-created file1 file). Then, the modifier ':h' strips off the file at the end, leaving just the directory name. I use this all the time to quickly hop into a folder of a file I just looked at.

4.4. History Env Vars

A quick note on environment variables that affect the history kept.

Type this in and try and figure out what's going on.

```
25$ bash
26$ HISTTIMEFORMAT="%d/%m/%y %T "
27$ history | tail
28$ HISTTIMEFORMAT="%d/%m/%y "
29$ history | tail
30$ HISTSIZE=2
31$ ls
32$ pwd
33$ history | tail
34$ ~/.bash_history | tail
35$ exit
36$ history
37$ ~/.bash_history | tail
```

By default 500 commands are retained in your shell's history. To change this setting, the 'HISTSIZE' variable must be set to the number you want.

There is also a HISTFILESIZE variable which determines the size of the history file itself. I did not get you to reduce the size of this to '1', as it would have wiped your history file and you might have got cross with me! But you can play with it if you want.

Finally, the 'HISTTIMEFORMAT' determines what time format should be shown with the bash history item. By default it's unset, so I usually set mine everywhere to be '%d/%m/%y %T '.

You should have noticed that the '~/.bash_history' file did not get updated with the 'ls' and 'pwd' commands until bash exited. It's a common source of confusion that the bash history is not written out until you exit. If your terminal connection freezes, your history from that session may never be written out. This frequently annoys me!

4.5. History Control

There's another history-controlling environment variable worth understanding:

```
38$ HISTCONTROL=ignoredups:ignorespace
39$ ls
40$ ls
41$  pwd    # <- note the space before the 'pwd'
42$ pwd
43$ ls
44$ history | tail
```

Was the output of history what you expected? HISTCONTROL can determine what gets stored in your history. The directives are separated by colons. Here we use 'ignoredups' to tell history to ignore commands that are repeats of the last-recorded command. In the above input, the two consecutive 'ls'es are combined into one in the history. If you want to be really severe about your history, you can also use 'erasedups', which adds your latest command to the history, but then wipes all previous examples of the same command out of the history. What would this have done to the history output above?

'ignorespace' tells bash to not record commands that begin with a space, like the 'pwd' in the listing above.

4.6. CTRL-R

Bash offers you another means to use your history.

Hit CTRL and hold it down. Then hit the 'r' key. You should see this on your terminal:

```
(reverse-i-search)``:
```

Let go. Now type 'grep'. You should see a previous grep command. If you keep hitting CTRL+r you will cycle through all commands that had grep in them, most recent first.

If you want to cycle forward (if you hit CTRL+r too many times and go past the one you want (I do

this a lot)), hit CTRL+s.

4.6.1. What You Learned

- Where bash keeps a history of commands
- How to refer to previous commands
- How to re-run a previous command with simple adjustments
- How to pick out specific arguments from the previous command
- How to control the history output
- How to control the commands that are added to the history
- How to search through your history dynamically

4.6.2. What Next?

Next you will tie these things together in a series of miscellaneous tips that finish off this part.

4.6.3. Exercises

1) Remember to use one of the above practical tips every day until you don't think about using it. Then learn another one.

2) Read up on all the history shortcuts. Pick ones you think will be useful.

3) Amend your bash startup files to control history the way you want it.

4) Think about where your time goes at the command line (eg typing out directories or filenames) and research whether there is a way to speed it up.

TODO

Misc

Get a date on the script: (read)

```
----
[metcalfs@gueabetapp37 ~]$
function dateit {
   while read line
   do
     printf "$line"
     date '+ %m-%d-%Y %H:%M:%S'
   done
}
[metcalfs@gueabetapp37 ~]$ vmstat 1 | dateit
----


Directory of script
```

```
------------------

----
$(dirname ${BASH_SOURCE[0]})
----

http://stackoverflow.com/questions/59895/can-a-bash-script-tell-what-directory-its-
stored-in

    DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"


Is a useful one-liner which will give you the full directory name of the script no
matter where it is being called from

These will work as long as the last component of the path used to find the script is
not a symlink (directory links are OK). If you want to also resolve any links to the
script itself, you need a multi-line solution:

    SOURCE="${BASH_SOURCE[0]}"
    while [ -h "$SOURCE" ]; do # resolve $SOURCE until the file is no longer a symlink
      DIR="$( cd -P "$( dirname "$SOURCE" )" && pwd )"
      SOURCE="$(readlink "$SOURCE")"
      [[ $SOURCE != /* ]] && SOURCE="$DIR/$SOURCE" # if $SOURCE was a relative
symlink, we need to resolve it relative to the path where the symlink file was located
    done
    DIR="$( cd -P "$( dirname "$SOURCE" )" && pwd )"


:chapter: 4

== Advanced Bash

//ADVANCED

//file substitution <()
//traps!
//signals
//auto-completion
//https://google.github.io/styleguide/shell.xml
//getopts
//command
//dev/tcp
//key variables: TERM, USER, DISPLAY


Putting it all together:
https://github.com/ianmiell/cheapci/blob/master/cheapci
```