IAN MIELL

LEARN BASH THE HARD WAY

MASTER BASH USING THE ONLY METHOD THAT WORKS

Learn Bash the Hard Way

Master Bash Using The Only Method That Works

Ian Miell

This book is for sale at http://leanpub.com/learnbashthehardway

This version was published on 2018-01-01

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Ian Miell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #learnbashthehardway.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#learnbashthehardway

Contents

Foreword

Almost every day of my 20-year software career, I've had to work with bash in some way or other. Bash is so ubiquitous that we take it for granted that people know it, and under-valued as a skill because it's easy to 'get by' with it.

It's my argument that the software community is woefully under-served when it comes to learning about bash, and that mastering it pays massive dividends, and not just when using bash.

Either you are given:

- impenetrable man pages full of jargon that assumes you understand far more than you do
- one-liners to solve your particular problem, leaving you no better off the next time you want to do something
- bash 'guides' that are like extended man pages - theoretical, full of jargon, and quite hard to follow

All of the above is what this book tries to address.

If you've ever been confused by things like:

- the difference between `[` and `[[`
- the difference between globbing and regexes
- the difference between single or double quoting in bash
- what `` ` `` means
- what a subshell is
- your terminal 'going crazy'

then this book is for you.

It uses the 'Hard Way' method to ensure that you have to understand what's needed to be understood to read those impenetrable man pages and take your understanding deeper when you need to.

Enjoy!

Learn Bash the Hard Way

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.

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.

Why Learn Bash The 'Hard Way'?

The 'Hard Way' is a method that emphasises the process required to learn anything. You don't learn to ride a bike by reading about it, and you don't learn to cook by reading recipes. Books can help (this one hopefully does) but it's up to you to do the work.

This book shows you the path in small digestible pieces based on my decades of experience and tells you to *actually type out the code*. This is as important as riding a bike is to learning to ride a bike. Without the brain and the body working together, the knowledge does not get there.

If you follow this course, you will get an understanding of bash that can form the basis of mastery as you use it going forward.

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 subtleties and underpinning technologies of bash
- Explain everything. Plenty of time to go deeper and get all the nuances later if you need them

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.

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 like this then you are in bash:

```
3.2.57(1)-release
```

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.

Any explanatory text will assume you typed it out. Again, this is the Hard Way, and we use it because it works.

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. Eventually you will be on our own out there and will need to think.

Each section is self-contained, and must be followed in full. To help show you where you are, the shell command lines 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.

Structure

This book is structured into four parts:

Part I - Core Bash

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

Part II - Scripting Bash

Gets your bash scripting skills up to a proficient point.

Part III - Tips

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

Part IV - Advanced Bash

Building on the first three chapters, and introducing some more advanced features, this chapters takes your bash skills beyond the norm.

Part I - Core Bash

This part takes you through some fundamental concepts necessary to take your bash knowledge further.

In it we cover:

- What bash is
- Globbing
- Variables
- Functions
- Pipes and Redirects
- Basic bash Scripting

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.

> ### Note
>
> A binary is a file that contains the instructions for a program, ie it is a 'program' file, rather than a 'text' file, or an 'application' file (such as a Word document).

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 within one another!

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

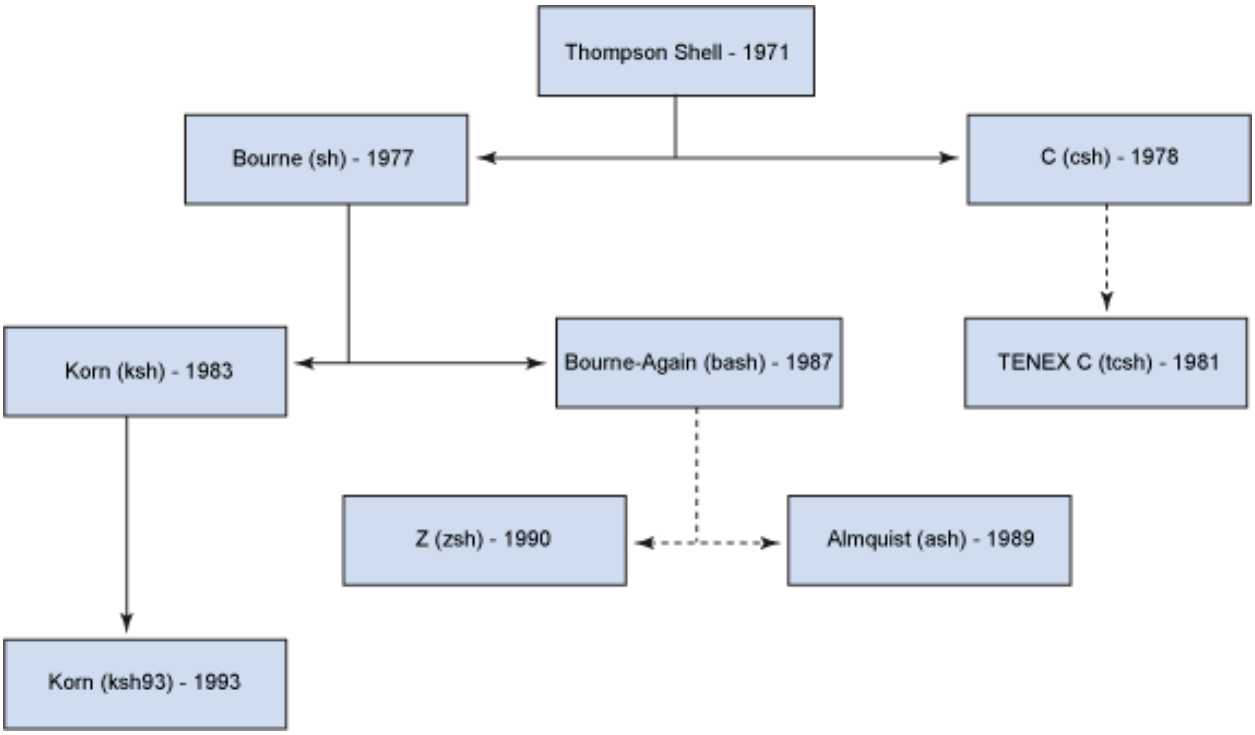
```
$ csh
% echo $dirstack
% exit
$
```

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 typing the echo commmand in when you are back in the bash shell at the end!)

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 unheard of to end up on servers that do not have bash!

What You Learned

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

What Next?

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

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`?

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.

> ## Note
>
> Do not panic if you don't know what regular expressions are. Regular expressions are patterns used to search for matching strings. Globs look similar and perform a similar function, but are not the same. That's the key point in the above paragraph.

Globbing

Type these commands into your terminal

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

- Line 1 above makes a new folder that should not exist already.
- Line 2 moves into that folder.
- Line 3 creates three files (file1,file2,file3).
- 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.

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 '*'
7 $ ls "*"
8 $ echo '*'
9 $ echo "*"
```

- Line 6 lists files matching the * character in single quotes
- Line 7 lists files matching the * character in double quotes
- Line 8 echoes the * character in single quotes
- Line 9 echoes 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!

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
11 $ ls file[a-z]
12 $ ls file[0-9]
```

- Line 10 list all the files that end in '1'
- Line 11 list all files that start with 'file' and end with a character from a to z
- Line 12 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 have a different significance depending on whether it is being treated as a glob or a regular expression.

```
13 $ rename -n 's/(.*)(.*)/new$1$2/' *
14 'file1' would be renamed to 'newfile1'
15 'file2' would be renamed to 'newfile2'
16 'file3' would be renamed to 'newfile3'
```

- Line 13 prints the files that would be renamed by the rename command if the `-n` flag were removed
- Lines 14-16 show the files that would be renamed

The first two * characters are treated as regular expressions, because they are not interpreted by the shell, but rather by the `rename` command. The reason they are not interpreted by the shell is because they are enclosed in quotes. The last * is treated as a glob by the shell, and expands to all the files in the local directory

> **Note**
>
> 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 more powerful extended globbing capabilities. Bash is one of the shells that offers extended globbing, which we do not cover here, as it would potentially confuse the reader further. Just be aware that more sophisticated globbing is possible.

Cleanup

Now clean up what you just did:

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

What You Learned

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

What Next?

Next up is another fundamental topic: variables.

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’).

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.

Basic Variables

Start by creating a variable and echoing it.

```
$ MYSTRING=astring
$ 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.

Variables and Quoting

Things get more interesting when you start quoting.

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

```
$ MYSENTENCE=A sentence
$ MYSENTENCE="A sentence"
$ 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  $ echo $MYSENTENCE
8  $ MYSENTENCE='A sentence with $MYSTRING in it'
9  $ echo $MYSENTENCE
```

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!

Type this out and see. As ever, make sure you think about the output you expect before you see it:

```
10 $ MYGLOB=*
11 $ echo $MYGLOB
12 $ MYGLOB="*"
13 $ echo $MYGLOB
14 $ MYGLOB='*'
15 $ echo $MYGLOB
```

Globs are not expanded when in either single or double quotes. Confusing isn't it?

Shell Variables

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

```
16 $ echo $PPID
17 $ PPID=nonsense
18 $ echo $PPID
```

- Line 16 - `PPID` is a special variable set by the bash shell. It contains the bash's parent process id.
- Line 17 - Try and set the `PPID` variable to something else.
- Line 18 - Output `PPID` again.

What happened there?

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

```
$ readonly MYVAR=astring
$ MYVAR=anotherstring
```

env

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

```
$ 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/g\
it/shutit:/space/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 of `env` will likely be different wherever you run it.

export

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

```
$ MYSTRING=astring
$ bash
$ echo $MYSTRING
$ exit
$ echo $MYSTRING
$ unset MYSTRING
$ echo $MYSTRING
$ export MYSTRING=anotherstring
$ bash
$ echo $MYSTRING
$ 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 'echoes' the new value 'anotherstring' to the terminal.

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

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:

```
$ bash --version
$ echo $BASH_VERSINFO
$ echo $BASH_VERSINFO[0]
$ echo ${BASH_VERSINFO[0]}
$ 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:

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

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

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

So all bash variables are 'really' arrays!

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

```
$ echo ${BASH_VERSINFO[1]}
$ echo ${BASH_VERSINFO[2]}
$ echo ${BASH_VERSINFO[3]}
$ echo ${BASH_VERSINFO[4]}
$ echo ${BASH_VERSINFO[5]}
$ echo ${BASH_VERSINFO[6]}
```

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

Associative Arrays

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

To check whether you have version 4+, run:

```
$ bash --version
```

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

```
$ declare -A MYAA=([one]=1 [two]=2 [three]=3)
$ MYAA[one]="1"
$ MYAA[two]="2"
$ echo $MYAA
$ echo ${MYAA[one]}
$ MYAA[one]="1"
$ WANT=two
$ 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.

What You Learned

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

What Next?

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

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.

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:

- A function
- An alias
- A program
- A builtin

This section covers these items, and the relationships between them.

Basic Functions

Start by creating a simple function

```
$ function myfunc {
        echo Hello World
}
$ 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.

Arguments

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

Predict the output of this, and then run it:

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

Can you explain the output? If not, you may want to read the previous pages!

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

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:

```
$ function myfunc {
        echo $myvar
}
$ myfunc
$ myvar="Hi from outside the function"
$ 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:

```
$ function myfunc {
        local myvar="Hi from inside the function"
        echo $myvar
}
$ myfunc
$ echo $myvar
$ 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.

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

Functions, Builtins, Aliases and Programs

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`!

```
21 $ builtin cd /tmp
22 $ cd -
23 $ builtin grep
24 $ 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 such 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`?

```
$ function cd() {
        echo 'No!'
}
$ cd /tmp
$ builtin cd /tmp
$ cd -
$ unset -f cd
$ cd /tmp
$ 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.

Now type this in:

```
$ declare -f
$ declare -F
```

If you want to know what functions are set in your environment, you 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 builtin or a separate binary?

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.

```
$ which grep
$ which cd
$ which builtin
$ 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:

```
$ alias cd=doesnotexist
$ alias
$ cd
$ unalias cd
$ cd /tmp
$ cd -
$ alias
```

And yes, you can alias alias.

What You Learned

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

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.

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!).

Pipes and redirects

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

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

Basic Redirects

Start off by creating a file:

```
$ mkdir lbthw_pipes_redirects
$ cd lbthw_pipes_redirects
$ echo "contents of file1" > file1
```

Basic pipes

Type this in:

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

> **Note**
>
> 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!

```
$ cat file2
```

What was the output of that?

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

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

Was that what you expected? If it was, you're doing well!

If it wasn't, then the answer is related to standard output and other kinds of output. We will explain this further below.

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'.

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

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

The file `/dev/null` is a special file created by Linux (and 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`.

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

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

Since standard output is pointed at the terminal at that time, standard error is also pointed at the terminal. From your point of view you see no difference, since by both standard output and standard error are pointed at the terminal anyway.

But when we try and redirect to standard error or standard output to files things get interesting, as you can change where they go. You saw this above when we redirected standard error to `/dev/null`.

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

```
$ command_does_not_exist 2>&1 > outfile
$ 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 (file descriptor '2') at whatever standard output (file descriptor '1') 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 from there on.

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
- the standard output points at the file 'outfile'.

In the second line (`command_does_not_exist > outfile 2>&1`), 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 file descriptor `2` (standard error) to whatever file descriptor `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!

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:

```
$ 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.

```
$ echo line1 > file3
$ echo line2 > file3
$ echo line3 >> file3
$ 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:

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

What You Learned

- What file redirection
- What pipes do
- The differences between standard output and standard error
- How to redirect standard output to the same location as standard error (and vice versa)
- How to redirect standard output or standard error (or both) to a file

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.

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'.

Scripts and Startups

This section considers two related subjects:

- shell scripts
- 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.

Shell scripts

```
$ mkdir -p lbthw_scripts_and_startups
$ 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:

```
$ echo '#!/bin/bash' > simple_script
$ 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:

```
$ ./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:

```
$ mkdir tmp
$ cd tmp
$ ../simple_script
$ cd ..
$ 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:

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

The chmod program changes the permissions (or 'modes' on a file), so that only certain users, or groups of users can read, write or execute (ie run as a program) a file.

> ### Note
>
> 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:

```
$ 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:

```
$ 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/sh\
utit:/space/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?

```
$ PATH=${PATH}:.
$ 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.

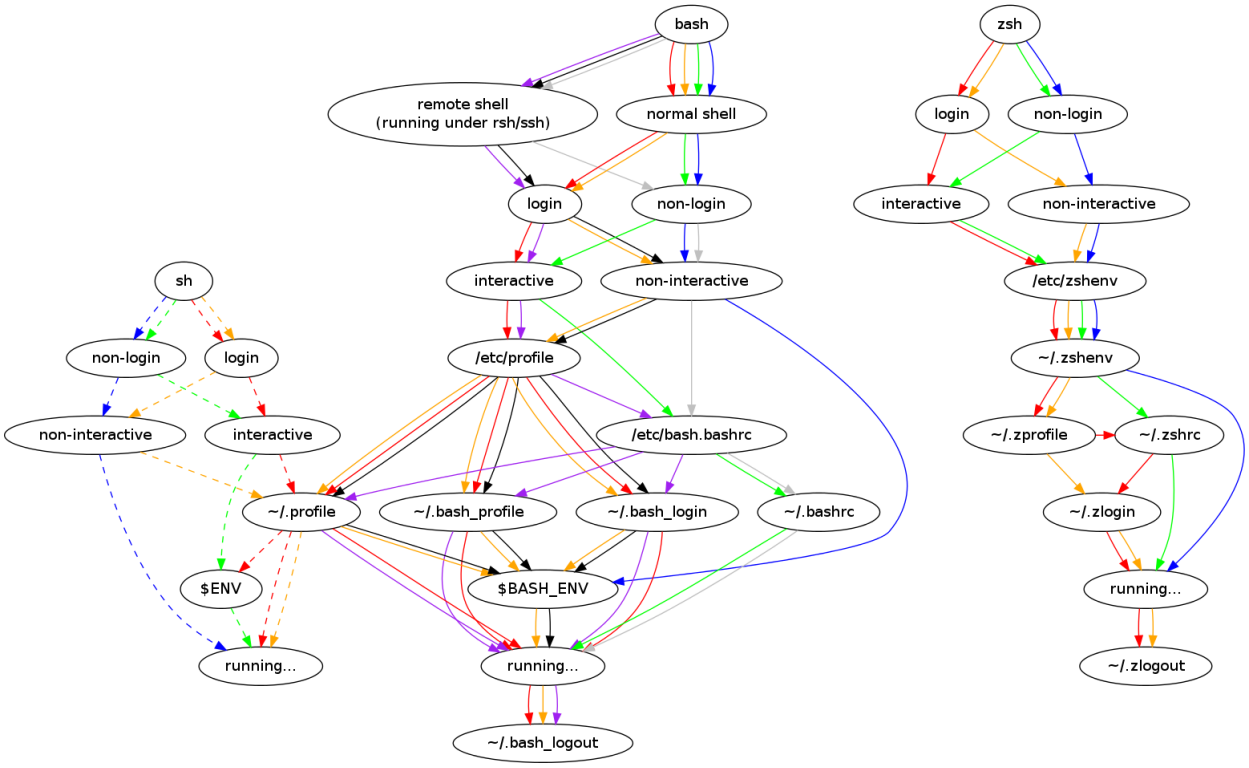
Startup Scripts

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.

Startup Explained

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

Have a look at this diagram:



Note

If the above diagram can't be easily viewed on your device, try a google search for 'shell startup diagram', or visit https://blog.flowblok.id.au/2013-02/shell-startup-scripts.html

Yes, this can be confusing.

The diagram shows the startup script order for different shells in different contexts. Each context is shown by following a separate colour through the diagram.

We are going to follow (from the top) the path from the 'bash' bubble, and ignore the 'zsh' and 'sh' paths, 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 (eg whether the shell is 'interactive' or not), or runs a script if the colour has already been determined from these decisions.

We'll walk through this below, which should make things clearer.

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`.

```
$ MYVAR=Hello
$ echo 'echo $MYVAR' > simple_echo
$ chmod +x simple_echo
$ ./simple_echo
$ source simple_echo
```

I'm sure you can figure out from that that `source` runs the script from within the same shell context. If you didn't see that, think carefully about what each command is doing.

Note

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:

```
$ cd ..
$ rm -rf lbthw_scripts_and_startups
$ unset MYVAR
```

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

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.

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).

Part II - Scripting Bash

In this section I cover concepts and techniques that form the basis of bash scripting. These are the fundamental building blocks required to write and comprehend useful bash scripts, and forms the basis of the more advanced sections that follow.

In it we cover:

- Command substitution
- The concept of the ‘test’ in bash
- Looping in bash
- Exit codes and bash
- The `set` command

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.

Command Subsitution Example

An example may help illustrate. Type these commands:

```
$ hostname
$ echo 'My hostname is: $(hostname)'
$ echo "My hostname is: $(hostname)"
```

If those lines are placed in a script, it will output the hostname of the host the script is running on. This can make your script much more dynamic. You can set variables based on the output of commands, add debug, and so on, just as with any other programming language.

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:

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

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

The 'Dollar-Bracket' Method: `$()`

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 (line 6) and moved into it (line 7).

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 substituted in over the `$(ls ..)`. The 'words' returned are placed as the arguments to the `touch` command. The `touch` command creates a set of empty files, based on the list of the parent directory's contents.

The `touch` command outputs the files created. These files are substituted in over the entire `$(touch $(ls ..))` of the command.

The `echo` command takes the output of the above substitution, ie the list of all the files created.

So, in summary:

- the line outputs the list of files of the parent directory
- 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 `$()` inside another command wrapped inside a `$()` and bash substitute in the output of each command from the inside out for you in the appropriate order.

Now let's look at the equivalent code with backticks.

The 'Backtick' Method

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 simplicity of managing nesting. 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)!

Cleanup

Remove the left-over directory:

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

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

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.

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 example.

Tests

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

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

What Are Bash Tests?

Tests in bash are constructs that allow you to do *conditional expressions*. They use square brackets (ie `[` and `]`) to enclose what is being tested.

For example, the simplest tests might be:

```
$ [ false = true ]
$ echo $?
$ [ true = true ]
$ echo $?
```

> ### Note
>
> 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:

```
$ A=1
$ [ $A = 1 ]
$ echo $?
$ [ $A == 1 ]
$ echo $?
$ [ $A = 2 ]
$ 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.

What is `[`, Really?

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

```
$ which [
$ builtin [
```

and that `[` and `test` are synonymous:

```
$ which test
$ builtin test
```

> **Note**
>
> `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 `[`. The `[` is a *separate command* and spacing is how bash determines where one command ends and another begins.

Logic operators

What do you expect the output of this to be?

```
$ ( [ 1 = 1 ] || [ ! '0' = '0' ] ) && [ '2' = '2' ]
$ 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:

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

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

If you're not confused yet, you might be soon! If you are, try and re-read the above until you get it.

The `[[` Operator

The `[[` operator is very similar to the 'test' operator with *two* square brackets instead of one:

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

This confused me a lot for some time! What is the difference between `[` and `[[` if they produce such similar output?

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

```
$ unset DOESNOTEXIST
$ [ ${DOESNOTEXIST} = '' ]
$ echo $?
$ [[ ${DOESNOTEXIST} = '' ]]
$ echo $?
$ [ x${DOESNOTEXIST} = x ]
$ echo $?
```

The first command above should error because the variable `DOESNOTEXIST`... does not exist. So bash processes that variable in the next line, and ends up running:

```
1  [ = '' ]
```

which makes no sense to it, so it complains! It's expecting something on the left hand side of the empty quotes.

The fourth command above (which uses the double brackets `[[`) tolerates the fact that the variable does not exist, and treats it as the empty string. It therefore resolves to:

```
1  [ '' = '' ]
```

The sixth command acts as a workaround. By placing an `x` on both sides of the equation, the code ensures that *something* gets placed on the left hand side:

```
1  [ 'x' = 'x' ]
```

You can 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. And it doesn't seem to do any harm.

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.

> **Note**
>
> If you want to understand more, go to http://serverfault.com/questions/52034/what-is-the-difference-between-double-and-single-square-brackets-in-bash

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, usually in the bash man page.

Unary and Binary Operators

There are other shortcuts relating to `test` (and its variants) that it's worth knowing about. These take a single argument:

```
$ echo $PWD
$ [ -z "$PWD" ]
$ echo $?
$ unset DOESNOTEXIST
$ [ -z "$DOESNOTEXIST" ]
$ echo $?
$ [ -z ]
$ 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`:

```
$ mkdir lbthw_tmp_dir
$ touch lbthw_tmp_file
$ [ -a lbthw_tmp_file ]
$ echo $?
$ [ -d lbthw_tmp_file ]
$ echo $?
$ [ -a lbthw_tmp_dir ]
$ echo $?
$ [ -d lbthw_tmp_dir ]
$ echo $?
$ 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.

Types

Type-safety (if you're familiar with that concept 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:

```
$ [ 10 < 2 ]
$ echo $?
$ [ '10' < '2' ]
$ echo $?
$ [[ 10 < 2 ]]
$ echo $?
$ [[ '10' < '2' ]]
$ 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:

```
$ [ 10 -lt 2 ]
$ echo $?
$ [ 1 -lt 2 ]
$ echo $?
$ [ 10 -gt 1 ]
$ echo $?
$ [ 1 -eq 1 ]
$ echo $?
$ [ 1 -ne 1 ]
$ 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.

`if` statements

Now you understand tests, if statements will be easy!

Type this:

```
$ 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:

```
$ 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.

Bare `if` statements

It's easy to forget that the `if` statement in bash does not need angle brackets at all. If the code between the `then` and the `if` is a bash command, then it will trigger if the exit code of the command was 'true':

What will this output? No cheating!

```
$ if grep not_there /dev/null
then
    echo there
else
    echo not there
fi
```

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

What Next?

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

Exercises

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

2) Write a script to check whether key files and directories are in their correct place.

3) Use the `find` and `wc` commands to count the number of files on your system and perform different actions if the number is higher or lower than what you expect.

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.

`for` Loops

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

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

```
$ 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:

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

with a similar effect.

`while` and `until`

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

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

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

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

`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:

```
$ a=1
$ 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.

`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:

```
$ mkdir lbthw_loops
$ cd lbthw_loops
```

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

```
$ 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
$ chmod +x case.sh
```

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

```
$ ./case.sh -a
$ ./case.sh -b
$ ./case.sh -b "an argument"
$ ./case.sh -a -b -c
$ ./case.sh
```

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

Cleanup

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

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!

What Next?

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

Exercises

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

2) Write a while loop to check where a file exists every few seconds. When it does, break out of the loop with a message.

Exit Codes

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

What Is An Exit Code?

After 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, this is a similar concept to that.

To take a simple example, type this in:

```
$ ls
$ echo $?
$ doesnotexist
$ echo $?
```

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

Now that you've learned about funcions, commands and now a little about exit codes, you should be able to follow what is going on here:

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

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

Standard Exit Codes

There are guidelines for exit codes for those that want to follow standards. Be aware that not all programs follow these standards (`grep` is the most common example, as you will learn)!

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)

> **Note**
>
> 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.

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:

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

Did you expect that? grep finished successfully (there was no segmentation or memory fault, memory, it was not killed etc..) but no lines were matched, and it returned '1' as an exit code. So grep uses the exit code '0' to mean 'matched', and the exit code '1' to mean 'not matched'.

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

```
$ 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 nearly 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!

Setting Your Own Exit Code

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

Type this:

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

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 these contexts.

Two of the most important are used in the below listing. Try and figure out what they are from context:

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

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

Cleanup

Now clean up what you just did:

```
$ rm afile.txt
```

What You Learned

- What an exit code is
- Some standard exit codes and their significance
- Not all applications use exit codes in the same way
- How tests and exit codes work together
- Some special parameters

What Next?

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

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.

The `set` Builtin

When using bash it is very important to understand what options are, how to set them, 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.

Running `set`

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

Note

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

[a] https://en.wikipedia.org/wiki/Standardization
[b] https://en.wikipedia.org/wiki/IEEE_Computer_Society
[c] https://en.wikipedia.org/wiki/Operating_system

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

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

```
$ bash
$ set -o posix
$ set
$ 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!

> **Note**
>
> 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:

```
$ 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.

`set` VS `env`

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

```
$ set
$ env
```

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

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:

```
$ set -o errexit
$ 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 each command as it is being run. This is really useful for seeing what command was actually run if (for example) you are using variables within your commands. It also helps you see the order in which commands are bing run.

Type this is to see how this works in practice:

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

You should be able to explain to someone else what's going on at each line typed in, and what the output of the above means.

Flags With `set` Instead of Names

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

```
$ set -e
$ set -x
```

is the same as:

```
$ set -o errexit
$ 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

[... remainder of script ...]
```

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

The pipefail Option, Exit Codes and Pipelines

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

```
$ set -o pipefail
$ grep notthere afile.txt | xargs echo
$ echo $?
$ set +o pipefail
$ grep notthere afile.txt | xargs echo
$ 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:

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

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

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.

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.

Part III - Bash Features

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

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 the ‘end of file’ characters. They don’t form part of words, or necessarily print anything to the screen, but they are bytes 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

> **Note**
>
> The 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.

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 that don’t involve writing a character.

It’s easy to forget this, but not everything that is sent to the computer 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`.

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 `-e` is set by default anyway.

Here 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` with these flags 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'
```

`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 (or move my cursor along a few spaces), as the terminal uses the tab key (if hit twice in a row) 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?

This is how: 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`).

If you type this in a terminal at the bash prompt, the cursor will tab across the screen in the way you might have previously expected.

Now hit `CTRL+v` and then the tab key. What happened? What does this tell you about the `CTRL+v` combination?

This character would normally be represented like this if you are trying to explicitly show it in a text editor (for example): `^I`

The caret (`^`) 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
```

There are multiple characters represented in this way. You've just seen tab (technically a 'vertical tab') represented as `^I`, and backspace represented as `^H`. You may recognise these and others in this table:

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

Name	Hex	C-escape	CTRL-key	Description
CR	0x0D	\r	^M	Carriage return
ESC	0x0E	N/A	^[	Escape character

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

Carriage Returns vs Line Feeds

The most commonly-seen non-printable character is the carriage return.

Carriage returns and line feeds cause much confusion, but it doesn't take long to understand the difference and (more importantly) 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 the `\r` and `\n` characters to represent a new line.

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.

What will this output?

```
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. Here a very useful tool comes in: `hexdump`

Hexdump

Run this:

```
$ echo -e 'Bad magazine\rMad' | hexdump
$ 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:

```
$ echo 'JUST THESE CHARACTERS' | hexdump -c
$ 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.

Terminal Escape Codes

Run this:

```
$ echo -e '\033[?47h'
$ 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`. So you could rewrite the above as:

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

These ESC and left bracket 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:

```
$ 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.

Fun With Terminals

Finally, some (optional) fun which pulls together a few different things you've learned along the way. It includes a couple of things (like `trap` which we cover in the next section, so don't stress too much about understanding it).

Create this listing 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
```

What You Learned

- What terminal codes are
- What printable and non-printable characters are

- How to output any arbitrary bytes
- How to prevent the terminal from interpreting the character using `CTRL-v`
- The difference between `\n` and `\r\n`
- What terminal escape codes are

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.

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.

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.

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.

The PS1 Variable

Type this:

```
$ bash
$ PS1='My super prompt>>>>   '
$ ls
$ exit
```

As you'll remember, there are some 'shell variables' that are set within bash that are used for various purposes. One of these is `PS1`, which is the prompt you see after each command is completed.

The PS2 Variable

Now try this:

```
$ bash
$ PS2='I am in the middle of something!>>>  '
$ cat > /dev/null << END
some text
END
$ 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. By default, this is set to '`> `', which is why you see that as the prompt when you normally type the `cat` command above in.

PS3 and PS4

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

PS4 is the last one:

```
$ bash
$ PS4='> Value of PWD is: $PWD'
$ set -x
$ pwd
$ cd /tmp
$ ls $(pwd)
$ cd -
$ exit
```

In 'trace' mode PS4 is echoed before each line of trace output. Do you remember what trace mode is?

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

> **Note**
>
> 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!

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:

```
$ bash
$ PS1='\u@\H:\w \# \$ '
$ ls
$ exit
```

The table below may help you understand:

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.

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

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

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

PROMPT_COMMAND

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

```
$ bash
$ PROMPT_COMMAND='echo "Hello prompt $(date)"'
$ ls
$ 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!

What You Learned

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

What Next?

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

Cleanup

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

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.

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.

> **Note**
>
> 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.

Basic Here Docs

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

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

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:

```
$ function write_echoer_file {
        cat > $1 << END
#!/bin/bash
echo $1
END
        chmod +x $1
}
$ write_echoer_file echoer
$ ./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:

```
$ function write_echoer_file {
        cat > $1 << 'END'
#!/bin/bash
echo $1
END
        chmod +x $1
}
$ write_echoer_file echoer
$ ./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 typed 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):

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

What if `END` is part of the 'here doc'?

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

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

Try and see what happens if it is.

Here Strings

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

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

Cleanup

```
47 $ rm -f echoer afile.txt
```

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

What Next?

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

Exercises

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

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.

Bash and History

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

Have a look at it (assuming it exists and has not been overridden by some startup script):

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

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:

```
$ mkdir lbthw_history
$ cd !$
$ echo 'About bash history' > file1
$ echo 'Another file' > file2
$ grep About file1
$ !!
$ grep About file2
$ grep Another !$
$ rm file2
$ !e
$ !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-runs 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 !$`.

How to Learn Them

The history items above are enough to be going on with if you've not seen them before. There's little point listing them all as you'll likely forget them before you finish this book.

So 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.

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 carets (`^`) 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':

```
$ grep another file1 | wc -l
$ # Is that output correct? I want to check the file by eye:
$ 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 the arguments start with '1':

```
$ grep another file1
$ 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 indicating 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:

```
$ LGTHWDIR=$(PWD)
$ cd /tmp
$ cat ${LGTHWDIR}/file1
$ 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.

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:

```
$ bash
$ HISTTIMEFORMAT="%d/%m/%y %T "
$ history | tail
$ HISTTIMEFORMAT="%d/%m/%y "
$ history | tail
$ HISTSIZE=2
$ ls
$ pwd
$ history | tail
$ ~/.bash_history | tail
$ exit
$ history
$ ~/.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!

History Control

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

```
$ HISTCONTROL=ignoredups:ignorespace
$ ls
$ ls
$  pwd    # <- note the space before the 'pwd'
$ pwd
$ ls
$ 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` commands 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.

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`.

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

What Next?

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

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.

Bash in Practice

So far we've been learning about bash in relatively abstract ways.

In this section you're going to see many of the ideas you've learned put together in more realistic contexts, so you can get a flavour for what bash can do for you on a day-to-day basis.

You can easily skip this section if you want as nothing here is new, but following this can embed some concepts and keep your motivation going before the final part!

Output With Time

Frequently, I want to get the output of a command along with the time. Let's say I'm running vmstat while a server is having problems, and I want to ensure that I've logged the time that vmstat relates to. Type this in:

```
$ function dateit() {
   while read line
   do
     echo "$line $(date '+ %m-%d-%Y %H:%M:%S')"
   done
}
$ vmstat 1 | dateit
```

> **Note**
>
> `vmstat` is a program available on most Linux flavours that gives you a view of what the system resource usage looks like. It is not available on Mac OSes. If vmstat does not exist for you, then replace with `top -s 2` or a similar command.

You will see the date appended to each line in the output. Experiment with the function to place the date before the line, or even on a separate line. See also the exercises below.

Where Am I?

You may be familiar with the `pwd` builtin, which gives you your current working directory (cwd). Similarly, there is an environment variable (`PWD`) that bash normally sets that stores the cwd.

```
$ pwd
$ echo $PWD
```

Very often in scripts, you will want to know where the cwd of the process is.

But also (very often) you will want to know where *the script you are running* is located *from within the script*.

For example, if you are running a script that is found in your `PATH` (ie not in your local folder), and you want to refer to another file relative to that script from within that script, then you will need to know where that script is located.

```
$ cat > /tmp/lbthwscript.sh << 'EOF'
echo My pwd is: $PWD
echo I am running in: $(dirname ${BASH_SOURCE[0]})
EOF
$ chmod +x /tmp/lbthwscript.sh
$ /tmp/lbthwscript.sh
```

Have a play with this function to see what it does.

What happens if you `cd` to `/tmp` and run it from there? Do you get an absolute path in the second line? What can you do about this?

Generic Extract Function

There are a bewildering number of archiving tools. To name a few of the most popular:

- tar
- zip
- bzip
- gzip
- compress

If you spend any time dealing with these files, then this is a candidate for a time-saving function to put into your startup files.

```
$ function extract() {
  if [ -z "$1" ]
  then
    echo "Usage: extract <file_name>.<zip|rar|bz2|gz|tar|tbz2|tgz|Z|7z|xz|ex|tar\
.bz2|tar.gz|tar.xz>"
  else
    if [ -f $1 ] ; then
       case $1 in
         *.7z)        7z x $1        ;;
         *.bz2)       bunzip2 $1     ;;
         *.exe)       cabextract $1  ;;
         *.gz)        gunzip $1      ;;
         *.tar.bz2)   tar xvjf $1    ;;
         *.tar.gz)    tar xvzf $1    ;;
         *.tar.xz)    tar xvJf $1    ;;
         *.tar)       tar xvf $1     ;;
         *.tbz2)      tar xvjf $1    ;;
         *.tgz)       tar xvzf $1    ;;
         *.Z)         uncompress $1  ;;
         *.xz)        unxz $1        ;;
         *.lzma)      unlzma $1      ;;
         *.rar)       unrar x -ad $1 ;;
         *.zip)       unzip $1       ;;
         *)           echo "extract: '$1' - unknown archive method" ;;
       esac
    else
        echo "$1 - file does not exist"
    fi
  fi
}
$ mkdir lbthwmisc
$ cd lbthwmisc
$ touch a b c
$ tar cvfz test.tgz a b c
$ rm a b c
$ extract test.tgz
```

Explain what each line does.

Output Absolute File Path

Quite often I want to give co-workers an absolute reference on a server to a file that I am looking at. One way to do this is to cut and paste the output of `pwd`, add a `/` to it, and then add the filename I want to share.

This takes a few seconds to achieve, and since it happens regularly, it's a great candidate for a time-saving function:

```
$ function sharefiles() {
      for file in $(ls "$@"); do
              echo -n $(pwd)
              [[ $(pwd) != "/" ]] && echo -n /
              echo $file
      done
}
$ sharefiles
```

Saving time by writing a function is often a great idea, but deciding what is worth automating is a non-trivial task.

Here are some things you want to think about when deciding what to automate:

- How often do you perform the task?
- How much effort is it to automate (it's easy to under-estimate this!)
- Will the automation require effort to maintain?
- Do you always perform the task on the same machine?
- Do you control that machine?

My experience is that the effects of automation can be very powerful, but the above factors can also limit the return on investment.

Think about what you could automate today (see exercises)!

Cleanup

```
$ cd ..
$ rm -rf lbthwmisc
$ rm /tmp/lbthwscript.sh
```

Exercises

1) Look at your history to work out what you do most often at the terminal. Write a function to make these tasks quicker.

2) Change the dateit function so that it outputs the hostname, username of the running user, and the time to millisecond granularity.

3) Extend the 'Where Am I?' function to handle symbolic links. If you don't know what symbolic links are, research them!

4) Extend the archive script to handle files that do not have the appropriate suffix. Hint: you may want to research the `file` command to achieve this.

Part IV - Advanced Bash

In this section you will learn about some more advanced bash topics.

In it, we cover:

- Trapping signals
- Debugging techniques, and making your scripts more robust
- String manipulation

Finally, you will look at a real-world bash program that implements a simple continuous integration process, and see how the techniques learned in this course can be applied.

Traps and Signals

Signals are a fundamental part of Linux processes. Signals (as the name suggests) are a way for simple messages to be sent to processes.

Triggering signals

Any easy way to trigger a signal is one you will likely already have used.

Follow the instructions here:

```
$ sleep 999 # NOW HIT CTRL, HOLD IT DOWN AND THEN HIT C (CTRL-c)
$ echo $?
```

You should have got the output of a number over 130. You will of course remember that `$?` is a special variable that gives the exit code of the last-run command.

What you are less likely to have remembered is that exit codes over 128 indicate that a signal triggered the exit, and that to determine the signal number you take 128 away from the number you saw.

Bonus points if you did remember!

The signals are usually documented in the 'signal' man page.

```
$ man signal
$ man 7 signal
```

> **Note**
>
> `man` pages have different sections. `man man` will explain more if you're interested, but to get a specific section, you put the section number in the middle, as above. Find out what section 7 is by reading `man man`.

If the signals are not listed on the man pages on your machine, then google them!

Now figure out what the signal was, what the default action is for that signal and the signal name that is triggered when you git `CTRL-c`.

```
$ sleep 999 # NOW HIT CTRL, HOLD IT DOWN AND THEN HIT Z (CTRL-z)
$ echo $?
```

Which signal does `CTRL-z` trigger?

kill

Another way to send a signal to a process is another one you have also likely come across: the `kill` command.

The `kill` command is misnamed, because it needn't be used to terminate a process. By default, it sends the signal 15 (`TERM`), which (similar to 2) usually has the effect of terminating the program, but as the name suggests, is a stronger signal to terminate than `INT` (interrupt).

```
$ sleep 999 &
$ KILLPID=$(echo ${!})
$ echo ${KILLPID}
$ kill -2 ${KILLPID}
$ echo ${?}
$ wait ${KILLPID}
$ echo ${?}
```

> **Note**
>
> The curly braces are required with the `${!}` (which surprised me!). Bash interprets the '!' as being a history command (try it!). I'm not sure why (it works fine outside the `$()`), but it is an indication that it's perhaps wise to get into the habit of putting curly braces around your variable names in bash.

Can you explain why the echo after the kill outputs 0 and not 130?

Instead of `-2` in the above listing, you can use the signal name. Either `-INT` or `-SIGINT` will work. Try them.

Trapping Signals

Type out this first and follow the instruction:

```
$ while :; do sleep 5; done # NOW HIT CTRL-c
```

Now type out this one and follow the instruction:

```
$ mkdir -p lbthw_traps
$ cd lbthw_traps
$ cat > trap_exit.sh << END
#!/bin/bash
trap "echo trapped" INT
while :; do sleep 5; done
END
$ chmod +x trap_exit.sh
$ ./trap_exit.sh # NOW HIT CTRL-c
```

What's going on? In the second listing you used the `trap` builtin to inhibit the default response of the trap_exit process in the bash process and replace it with another response. In this case, the first argument to the `trap` builtin is evaluated and run as a command (`echo trapped`).

So how to get out of it and kill off the process?

```
$ # HIT CTRL-z
$ kill %1
```

Trap Exit

In addition to the normal signal name traps, there is some special ones.

Type this out:

```
$ cat > trap_exit.sh << END
#!/bin/bash
trap "echo trapped" EXIT
sleep 999 &
wait ${!}
END
$ trap_exit.sh &
$ TRAP_EXIT_PID=$(echo ${!})
$ kill -15 ${TRAP_EXIT_PID}
```

Which signal did we use there?

The `EXIT` trap catches the termination of the script and runs. Try it with `-2` as well.

Now run this:

```
$ trap_exit.sh &
$ TRAP_EXIT_PID=$(echo ${!})
$ kill -9 ${TRAP_EXIT_PID}
```

Some of the signals are not trap-able! Why do you think this is?

Experiment with some other signals to determine how `EXIT` handles them.

What is the name of the `-9` signal? Is this the default that the `kill` command uses?

A Note About Process Groups

You may have noticed that in the above script you used the `wait` command after putting the process in the background.

The wait command is a bash builtin that returns when the child processes of the bash process completes.

This illustrates a subtle point about signals. They act on the *currently running* process, and not on their children.

Repeat the above section, but rather than having:

```
sleep 999 &
wait ${!}
```

type:

```
sleep 999
```

What do you notice about the behaviour of the `EXIT` and `INT` signals?

How do you explain the fact that running this:

```
$ ./trap_exit.sh # HIT CTRL-c
```

works to kill the sleep process and output 'trapped', where sending the signal `-2` before did not?

The answer is that foregrounded processes are treated differently - they form part of a 'process group' that gets any signals received on the terminal.

If this seems complicated, just remember: `CTRL-c` kills all the processes 'going on' in the foreground of the terminal the `2`/`INT` signal, while `kill` sends a message to a specific process, which may or may not be running at the time.

If this seems complicated, just remember: signals can get complicated!

Cleanup

```
38 $ cd ..
39 $ rm -rf lbthw_traps
```

What You Learned

In this section you have learned:

- What a signal is
- What a trap is
- What the `kill` program does, and that it doesn't send `KILL` by default
- What an `INT` and `TERM` signal is
- How to trap exiting bash processes
- What a process group is, and its significance for signals

Exercises

1) Write a shell script that you can't escape from (the machine it runs on must not be overloaded as a result!) in the terminal

2) Try and escape from the shell script you created in 1)

3) Ask everyone you know if they can escape the shell script

4) If no-one can escape it, send it to the author :)

Debugging Bash Scripts

There are a few useful techniques worth knowing to help debug your bash scripts. While covering some of these here, you will also learn how to make your bash scripts more robust and less prone to failure.

Syntax Checking Options

Start by creating this simple script:

```
$ mkdir -p lbthw_debugging
$ cd lbthw_debugging
$ cat > debug_script.sh << 'END'
#!/bin/bash
A=some value
echo "${A}
echo "${B}"
END
```

Now run it with the -n flag. This flag only parses the script, rather than running it. It's useful for detecting basic syntax errors.

```
$ bash -n debug_script.sh
```

You can see it's broken. Fix it. Then run it:

```
$ bash debug_script.sh
```

Now run with `-v` to see the verbose output.

```
$ bash -v debug_script.sh
```

Try tracing to see more details about what's going on. Each statement gets a new line.

```
$ bash -x debug_script.sh
```

Using these flags together can help debug scripts where there is an elementary error, or even just working out what's going on when a script runs. I used it only yesterday to figure out why a systemctl service wasn't running or logging.

Managing Variables

Variables are a core part of most serious bash scripts (and even one-liners!), so managing them is another important way to reduce the possibility of your script breaking.

Change your script to add the 'set' line immediately after the first line and see what happens.

```
#!/bin/bash
set -o nounset
A="some value"
echo "${A}
echo "${B}"
```

Now research what the `nounset` option does. Which `set` flag does this correspond to?

Without running this, try and figure out what this script will do. Will it run?

```
#!/bin/bash
set -o nounset
A="some value"
B=
echo "${A}"
echo "${B}"
```

I always set `nounset` on my scripts as a habit. It can catch many problems before they become serious.

Tracing Variables

If you are working with a particularly complex script, then you can get to the point where you are unsure what happened to a variable.

Try running this script and see what happens:

```
#!/bin/bash
set -o nounset
trap 'echo "${BASH_SOURCE}>A \$A= \"${A}\" ${LINENO}"' DEBUG
declare -t A="some value"
B=
echo "${A}"
A="another value"
echo "${A}"
echo "${B}"
```

Now fix it.

Can you work out what is going on? You may need to refer to the bash man page, and make sure you understand quoting in bash properly.

The `-t` flag to declare 'traps' when the variable is referenced or assigned to.

Profiling Bash Scripts

Returning to the `xtrace` (or `set -x`) flag, we can exploit its use of a PS variable to implement the profiling of a script:

```
#!/bin/bash
set -o nounset
set -o xtrace
declare A="some value"
PS4='$(date "+%s%N")'
B=
echo "${A}"
A="another value"
echo "${A}"
echo "${B}"
ls
pwd
curl -q bbc.co.uk
```

Note

If you are on a Mac, then you might only get second-level granularity on the date!

Shellcheck

Finally, here is a very useful tip for understanding bash more deeply and improving any bash scripts you come across. Shellcheck is a website (http://www.shellcheck.net/) and a package that gives you advice to help fix and improve your shell scripts. Very often, its advice has prompted me to research more deeply and understand bash better.

Here is some example output from a script I found on my laptop:

```
$ shellcheck shrinkpdf.sh
In shrinkpdf.sh line 44:
          -dColorImageResolution=$3             \
                                 ^-- SC2086: Double quote to prevent globbing an\
d word splitting.


In shrinkpdf.sh line 46:
          -dGrayImageResolution=$3              \
                                ^-- SC2086: Double quote to prevent globbing and\
 word splitting.


In shrinkpdf.sh line 48:
          -dMonoImageResolution=$3              \
                                ^-- SC2086: Double quote to prevent globbing and\
 word splitting.


In shrinkpdf.sh line 57:
        if [ ! -f "$1" -o ! -f "$2" ]; then
                        ^-- SC2166: Prefer [ p ] || [ q ] as [ p -o q ] is not we\
ll defined.


In shrinkpdf.sh line 60:
        ISIZE="$(echo $(wc -c "$1") | cut -f1 -d\ )"
                     ^-- SC2046: Quote this to prevent word splitting.
                     ^-- SC2005: Useless echo? Instead of 'echo $(cmd)', just u\
se 'cmd'.


In shrinkpdf.sh line 61:
```

```
        OSIZE="$(echo $(wc -c "$2") | cut -f1 -d\ )"
                      ^-- SC2046: Quote this to prevent word splitting.
                      ^-- SC2005: Useless echo? Instead of 'echo $(cmd)', just u\
se 'cmd'.
```

The most common reminders are regarding potential quoting issues, but you can see other useful tips in the above output, such as preferred arguments to the `test` construct, and advice on 'useless' `echo`s.

Cleanup

To clean up the above work:

```
$ cd ..
$ rm -rf lbthw_debugging
```

What You Learned

In this section, you learned:

- bash flags useful for debugging
- how to use traps and declare to trace the use of variables
- how to make your scripts more robust with nounset
- how to use shelltrace to help you reduce the risk of your scripts failing

Exercises

1) Find a large bash script on a social coding site such as GitHub, and run shellcheck over it. Contribute back any improvements you find.

String Manipulation

Since so much of working in bash is related to files and strings of text, the ability to manipulate strings is valuable.

While tools such as `sed`, `awk`, `perl` (and many many others) are well worth learning, in this section I want to show you what is possible in bash - and it may be more than you think!

String Length

One of the most common requirements when working with strings is to determine length:

```
$ mkdir lbthw_strings
$ cd lbthw_strings
$ A='12345678901234567890'
$ echo "${#A}"
$ echo "$#A"
```

Why did the second one not 'work'?

String Editing

Bash provides a way to extract a substring from a string. The following example expains how to parse *n* characters starting from a particular position.

Work out what's going on here. You may need to consult the manual:

```
$ echo ${A:2}
$ echo ${A:2:3}
```

You can replace sections of scripts using search and replace. The first part enclosed in '/' signs represents what's searched for, and the second what is replaced:

```
$ echo "${A/234/432}"
$ echo "${A//234/432}"
```

What's going on in the second command above? How does it differ from the first?

Extglobs and removing text

A more advanced means of working with strings is possible by using bash's extglobs functionality.

A word of warning here: although this functionality is useful to know, it is arguably less useful than learning the programs `sed` and `perl` for this purpose. It's also quite confusing to have this extra type of glob syntax to learn in addition to regular expressions (and there's even flavours of thosei too!).

```
10 shopt -s extglob
11 A="12345678901234567890"
12 B="  ${A}  "
```

You've ensured that the `extglob` option is on, and created a new variable `B`, which is `A` with two spaces in front and behind. The pipe (|) character is used in the echo output to show where the spaces are.

```
13 echo "B      |${B}|"
14 echo "B#+( ) |${B#+( )}|"
15 echo "B#?( ) |${B#?( )}|"
16 echo "B#*( ) |${B#\*( )}|"
17 echo "B##+( )|${B##+( )}|"
18 echo "B###*( )|${B###*( )}|"
19 echo "B##?( )|${B##?( )}|"
```

Using the bash man page, and experimenting, can you figure out the differences between the above extglobs? They are:

- `?( )`
- `+( )`
- `*( )`

Now try `%` instead of `#` above. What happens?

One potentially handy application of this is when having to remove leading zeroes from dates:

```
20 TODAY=$(date +%j)
21 TODAY=${TODAY##+(0)}
```

Remember: `#` is to the left, and `%` is to the right on a (US) keyboard. Or 'hash' is before 'per cent' in the alphabet.

Quoting Hell

Quoting - as I'm sure you've seen - can get very complicated in bash very quickly.

Try this:

```
22 $ echo 'I really want to echo $HOME and I can't avoid it'
```

Uh-oh. You're now stuck.

Can you see why?

Hit `CTRL-c` to get out of it.

So you might think to try this:

```
23 $ echo "I really want to echo $HOME and I can't avoid it"
```

but this time the `$HOME` variable is evaluated and the output is not what is wanted.

Try this:

```
24 $ echo 'I really want to echo $HOME and I can'"'"'t avoid it'
```

That works. Remember this trick as it can save you a lot of time!

Cleanup

```
25 $ cd ..
26 $ rm -rf lbthw_strings
```

What You Learned

This section taught you about string management using pure bash. This is a relatively rare field to cover, but worth having seen in case you come across it. You learned:

- How to edit strings using pure bash
- How to determine string length
- What extglobs are and how you can use them
- How to avoid common quoting problems

Exercises

1) Learn how to do all of the above things in sed too. This will take some research and time.

2) Learn how to do all of the above in perl. This will also take some research and time!

3) Construct a useful echo that has a double-quoted string with a single-quoted string inside, eg one that outputs:

He said 'I thought she'd said "bash was easy when $s are involved" but that can't be true!'

Example Bash Script

To finish off the course, we're going to look at a slightly larger bash project to demonstrate the techniques learned in this course in a more pracitcal context.

Cheapci

`cheapci` is a bash script I wrote in an effort to understand what Jenkins was doing, and improve my bash.

The latest version available here: https://github.com/ianmiell/cheapci

I reproduce a modified cut of the code here with annotations. As with every listing in this course, I recommend typing the code (minus the comments if you prefer!) out to get a feel for the flow of the code.

Annotated code

Here is the code with annotations. Not every line is annotated, I just draw attention to the bits you've learned during the course:

```
#!/bin/bash

# Options are placed right at the top.
set -o pipefail
set -o errunset

# Set up defaults to sensible values.
FORCE=0
VERBOSE=0
REPO=""
EMAIL=""
NAME="ci"
TEST_DIR="."
TEST_COMMAND="./test.sh"
MAIL_CMD="mail"
MAIL_CMD_ATTACH_FLAG="-A"
MAIL_CMD_RECIPIENTS_FLAG="-t"
```

```
18 MAIL_CMD_SUBJECT_FLAG="-s"
19 PRE_SCRIPT="/bin/true"
20 POST_SCRIPT="/bin/true"
21 TIMEOUT_S=86400
22
23 # Set up the 'PAGER' variable, defaulting to 'more' in case it is unset in the
24 # environment.
25 PAGER=${PAGER:more}
26
27 # 'show_help' is a function that sends useful help to the standard output.
28 # I put it in a function in case it might be called from more than one place
29 # in the script.
30 function show_help() {
31         cat > /dev/stdout << END
32 ${0} -r <repo> -l <local_checkout> [-q <pre-script>] [-w <post-script>]
33    [-m <email>] [-a <mail command>] [-t <mail command attach flag>]
34    [-s <mail command subject flag] [-e <recipients flag>] [-n name] [-d <dir>]
35    [-c <command>] [-f] [-v] [-h]
36
37 REQUIRED ARGS:
38 -r - git repository, eg https://github.com/myname/myproj (required)
39 -l - local checkout of code (that gets updated to determine whether a run is nee\
40 ded) (required)
41
42 OPTIONAL ARGS:
43 -q - script to run just before actually performing test (default ${PRE_SCRIPT})
44 -w - script to run just after actually performing test (default ${POST_SCRIPT})
45 -m - email address to send to using "mail" command (default logs to stdout)
46 -a - mail command to use (default=${MAIL_CMD})
47 -n - name for ci (unique, must be a valid directory name), eg myproj (default=${\
48 NAME})
49 -d - directory within repository to navigate to (default=${TEST_DIR})
50 -c - test command to run from -d directory (default=${TEST_COMMAND})
51 -t - attach argument flag for mail command (default=${MAIL_CMD_ATTACH_FLAG}, emp\
52 ty string means no-attach)
53 -s - subject flag for mail command (default=${MAIL_CMD_RECIPIENTS_FLAG})
54 -e - recipients flag (default=${MAIL_CMD_RECIPIENTS_FLAG}, empty string means no\
55  flag needed)
56 -f - force a run even if repo has no updates (default off)
57 -v - verbose logging (default off)
58 -i - timeout in seconds (default 86400, ie one day, does KILL one hour after tha\
59 t)
```

```
-h - show help

EXAMPLES

- "Clone -r https://github.com/ianmiell/shutit.git if a git pull on /space/git/s\
hutit indicates there's been an update.
  Then navigate to test, run ./test.sh and mail ian.miell@gmail.com if there are\
 any issues"

  ./cheapci -r https://github.com/ianmiell/shutit.git -l /space/git/shutit -d te\
st -c ./test.sh -m ian.miell@gmail.com


- "Run this continuously in a crontab."

  Crontab line:

  * * * * * cd /path/to/cheapci && ./cheapci -r https://github.com/ianmiell/shut\
it.git -l /space/git/shutit -d test -c ./test.sh -m ian.miell@gmail.com
END
}

# Next we gather options that were passed in from the command line. Do you know
# what the ':'s and '?' symbols mean below?
while getopts "h?vfm:n:d:r:l:c:a:q:w:t:e:s:" opt
do
        case "$opt" in
        h|\?)
                show_help
                exit 0
                ;;
        v) VERBOSE=1 ;;
        f) FORCE=1 ;;
        r) REPO=$OPTARG ;;
        m) EMAIL=$OPTARG ;;
        n) NAME=$OPTARG ;;
        d) TEST_DIR=$OPTARG ;;
        l) LOCAL_CHECKOUT=$OPTARG ;;
        c) TEST_COMMAND=$OPTARG ;;
        a) MAIL_CMD=$OPTARG ;;
        q) PRE_SCRIPT=$OPTARG ;;
        w) POST_SCRIPT=$OPTARG ;;
```

```
102         a) MAIL_CMD=$OPTARG ;;
103         t) MAIL_CMD_ATTACH_FLAG=$OPTARG ;;
104         e) MAIL_CMD_RECIPIENTS_FLAG=$OPTARG ;;
105         s) MAIL_CMD_SUBJECT_FLAG=$OPTARG ;;
106         i) TIMEOUT_S=$OPTARG ;;
107         esac
108 done
109
110 # This line 'shifts' all the arguments off the command. The 'OPTIND' variable
111 # contains the number of options found by getopts.
112 shift "$((OPTIND-1))"
113
114 # We require that the git REPO variable is set up by this point, so if it is
115 # not, then we show the help and exit with a non-zero code to show that the
116 # run did not complete successfully.
117 if [[ ${REPO} = "" ]]
118 then
119         show_help
120         exit 1
121 fi
122
123 # Rather than set xtrace on every time, we set it on only if the verbose flag
124 # was used.
125 if [[ ${VERBOSE} -gt 0 ]]
126 then
127         set -x
128 fi
129
130 # More variables, this time derived from the optional values.
131 # Create variables for items that will be re-used rather than using 'magic
132 # values'.
133 BUILD_DIR_BASE="/tmp/${NAME}"
134 BUILD_DIR="${BUILD_DIR_BASE}/${NAME}_builddir"
135 mkdir -p "${BUILD_DIR}"
136 # Use of the RANDOM variable to create a log file hopefully unique to this run.
137 LOG_FILE="${BUILD_DIR}/${NAME}_build_${RANDOM}.log.txt"
138 BUILD_LOG_FILE="${BUILD_DIR}/${NAME}_build.log.txt"
139 # Create a lock file based on the name given
140 LOCK_FILE="${BUILD_DIR}/${NAME}_ci.lck"
141
142 # Create a generic cleanup function in case it is needed laster
143 function cleanup() {
```

```
144         rm -rf "${BUILD_DIR}"
145         rm -f "${LOCK_FILE}"
146         # get rid of /tmp detritus, leaving anything accessed 2 days ago+
147         find "${BUILD_DIR_BASE}"/* -type d -atime +1 | rm -rf
148         echo "cleanup done"
149 }
150
151 # Trap specific signals and run cleanup
152 trap cleanup TERM INT QUIT
153
154 # Function to send mail. Note the use of the array log_file_arg in the mail
155 # command.
156 function send_mail() {
157         msg=${1}
158         if [[ ${LOG_FILE} != "" ]] && [[ ${MAIL_CMD_ATTACH_FLAG} != "" ]]
159         then
160                 log_file_arg=(${MAIL_CMD_ATTACH_FLAG} ${LOG_FILE})
161         fi
162         if [[ ${EMAIL} != "" ]] && [[ ${MAIL_CMD} != "" ]]
163         then
164                 echo "${msg}" | ${MAIL_CMD} "${MAIL_CMD_SUBJECT_FLAG}" "${msg}" \
165 "${log_file_arg[@]}" "${MAIL_CMD_RECIPIENTS_FLAG}" "${EMAIL}"
166         else
167                 echo "${msg}"
168         fi
169 }
170
171 # Output the date to the log file.
172 date 2>&1 | tee -a "${BUILD_LOG_FILE}"
173
174 # Use the -a test to determine whether this ci is currently running.
175 if [[ -a ${LOCK_FILE} ]]
176 then
177         echo "Already running" | tee -a "${BUILD_LOG_FILE}"
178         exit
179 else
180         touch "${LOCK_FILE}"
181         # Fetch changes from the remote servers.
182         pushd "${LOCAL_CHECKOUT}"
183         git fetch origin master 2>&1 | tee -a "${BUILD_LOG_FILE}"
184         # See if there are any incoming changes
185         updates=$(git log HEAD..origin/master --oneline | wc -l)
```

```
186         echo "Updates: ${updates}" | tee -a "${BUILD_LOG_FILE}"
187         if [[ ${updates} -gt 0 ]] || [[ ${FORCE} -gt 0 ]]
188         then
189                 touch "${LOG_FILE}"
190                 pushd "${LOCAL_CHECKOUT}"
191                 echo "Pulling" | tee -a "${LOG_FILE}"
192                 git pull origin master 2>&1 | tee -a "${LOG_FILE}"
193                 popd
194                 # This won't exist in a bit so no point pushd'ing
195                 pushd "${BUILD_DIR}"
196                 # Clone to NAME
197                 git clone "${REPO}" "${NAME}"
198                 popd
199                 ${PRE_SCRIPT} 2>&1 | tee -a "${LOG_FILE}"
200                 EXIT_CODE="${?}"
201                 if [[ ${EXIT_CODE} -ne 0 ]]
202                 then
203                         msg="ANGRY ${NAME} on $(hostname)"
204                 fi
205                 pushd "${BUILD_DIR}"/"${NAME}"/"${TEST_DIR}"
206                 timeout "${TIMEOUT_S}" "${TEST_COMMAND}" 2>&1 | tee -a "${LOG_FI\
207 LE}"
208                 EXIT_CODE=$?
209                 popd
210                 if [[ ${EXIT_CODE} -ne 0 ]]
211                 then
212                         if [[ ${EXIT_CODE} -eq 124 ]]
213                         then
214                                 msg="ANGRY (TIMEOUT) ${NAME} on $(hostname)"
215                         else
216                                 msg="ANGRY ${NAME} on $(hostname)"
217                         fi
218                 else
219                         msg="HAPPY ${NAME} on $(hostname)"
220                 fi
221                 ${POST_SCRIPT} 2>&1 | tee -a "${LOG_FILE}"
222                 EXIT_CODE=$?
223             if [[ ${EXIT_CODE} -ne 0 ]]
224                 then
225                         msg="ANGRY ${NAME} on $(hostname)"
226                 fi
227                 send_mail "${msg}"
```

```
228         fi
229         cleanup
230 fi
```

If you have been following the course carefully, you will spot some improvements that could be made. See the exercises section for what to do if that is the case!

What You Learned

You won't have learned anything specifically new in this section, but I hope it has been made clear that bash can be used for more than just 'one-liners'.

Cleanup

You should know the drill by now.

Exercises

1) Find improvements to `cheapci` and submit them as pull requests. If you're not familiar with the pull request process, then create an account on GitHub and suggest your change by adding an issue and filling out the form.

Finished!

Well done! You've finished the course.

The course's aims were to give you an understanding of key concepts of bash so that you are:

- aware of what bash can do
- develop your understanding of central bash concepts

If I had to put it in a sentence, I'd say that this course should give you the ability to never be surprised or helpless in the face of bash you come across.

I hope that's what you got from the course!

I welcome feedback and comment to ian.miell@gmail.com