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 2019-09-25

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.

© 2017 - 2019 Ian Miell

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 . . . . . **i**

Learn Bash the Hard Way . . . . . **ii**
- Introduction ii
- Structure v

Part I - Core Bash **1**
- What is Bash? 2
- Unpicking the Shell: Globbing and Quoting 5
- Variables in Bash 11
- Functions in Bash 17
- Pipes and redirects 22
- Scripts and Startups 28

Part II - Scripting Bash **35**
- Command Substitution and Evaluation 36
- Exit Codes 39
- Tests 44
- Loops 52
- The `set` Builtin 56
- Process Substitution 61
- Subshells 65
- Internal Field Separator 70

Part III - Bash Features **74**
- Readline 75
- Terminal Codes and Non-Standard Characters 81
- The Prompt 89
- Here Documents 93
- History 97
- Bash in Practice 103

Part IV - Advanced Bash **108**
- Job Control 109
- Traps and Signals 113

Advanced Variables 118
String Manipulation 121
Debugging Bash Scripts 126
Autocomplete 131
Example Bash Script 137
Finished! 144

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.

Introduction

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 it sometimes doesn't work as you expect it to, or even why bash works after 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 in the future.

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 for yourself. I'm trying to prepare you for that day.

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 to leave no trace of your work.

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 chapter 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).

If you’re not sure what this means, then don’t worry. You only need to know that a shell program is a program that allows you to tell the computer what to do. In that way, it’s not much different to many other kinds of programming languages.

What makes bash different from some other languages is that it is a language designed to ‘glue’ together other programs.

In this section, you will learn a little about the history of bash and other related shells.

How Important is this Section?

This section is background and scene-setting material. It’s not essential to material in the rest of the book, but contains information it’s useful to be aware of when you read around the subject.

Other Shells

Other shells include:

- `sh`
- `ash`
- `dash`
- `ksh`
- `csh`
- `tcsh`
- `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 tcsh from within your bash terminal. Note that you get a different prompt (by default):

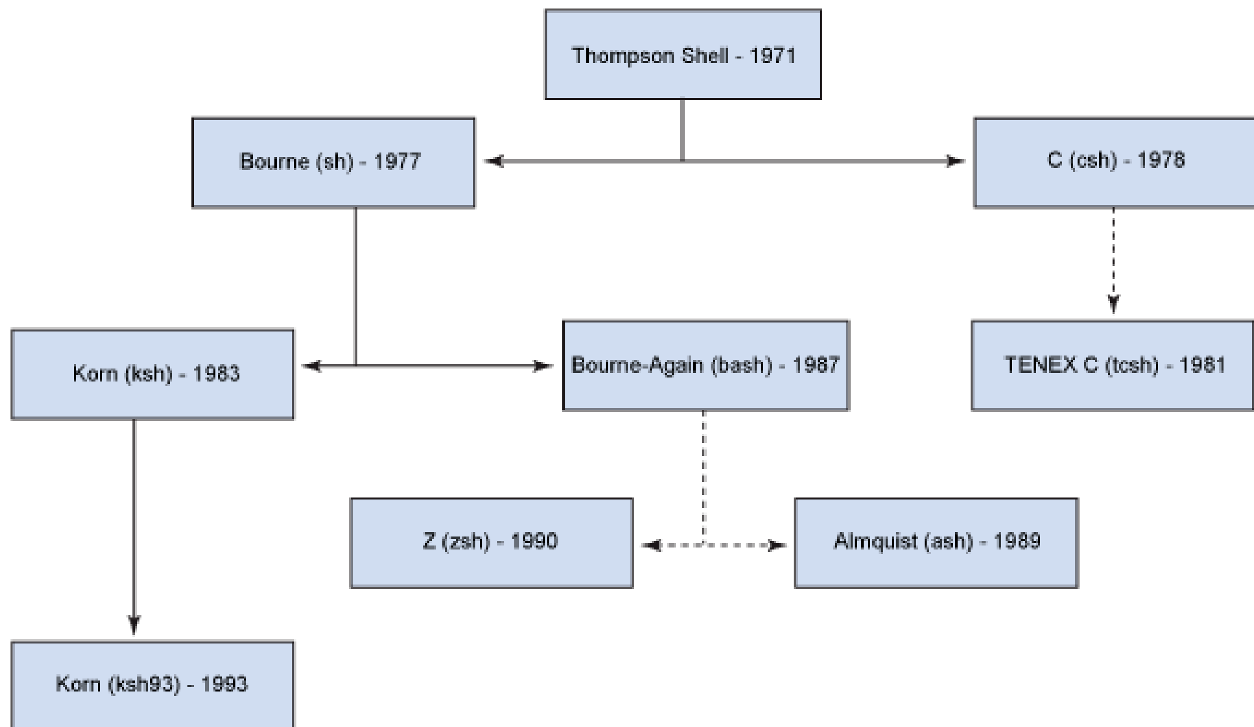
```
$ tcsh
% echo $dirstack
% exit
$
```

Typically, a tcsh 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 tcsh 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 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.

In this section you will learn:

- What globbing is
- The special globbing characters
- The difference between single and double quotes
- The difference between globbing and regular expressions

How Important is this Section?

Globbing and quoting are essential topics when using bash. It's rare to come across a set of commands or a script that doesn't depend on knowledge of them.

Globbing

Type these commands into your terminal

```
1 $ mkdir lbthw_glob
2 $ cd lbthw_glob
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

Dotfiles

Dotfiles are like normal files, except their name begins with a dot. Create some with `touch` and `mkdir`:

```
13  $ touch .adotfile
14  $ mkdir .adotfolder
15  $ touch .adotfolder/file1 .adotfolder/.adotfile
```

You've now created some dotfiles. If you run `ls`:

```
16  $ ls
```

those files don't show up. So these files are hidden from us in normal view. What if we try to use a `*` as a glob?

```
17  $ ls *
```

Same result. Those files are hidden. While this may seem (and sometimes is) annoying, having files that don't match even a `*` glob is very useful. Frequently you want to have a file that sits alongside other files but that is generally ignored. For example, you might write some code that reformats a set of text files in a folder, but you don't want to reformat a dotfile that contains information about what's in those text files.

Unfortunately, it can be annoying when you really do want to see *all* the files in a folder. To achieve this, type:

```
18  $ echo .*
```

This tells bash that you want to see all filenames in the current folder that begin with a `.` character. `ls .*` will give you a different output, but we will get to that in a moment.

Note

The . character has no special significance in a globbing context. It literally just means the dot character. If you know regular expressions already, then this can be very confusing, as .* means 'match everything'. We go over this again below.

You'll also get a couple of extra 'special' files shown that you may not have been aware of before. These are the single dot folder: ., and the double dot folder: ..

The single dot folder (.) is a special file that represents the folder that you are in. For example, if you type:

```
20 $ cd .
```

You will go nowhere! You've changed directory to the same folder that you are in.

The double dot folder (..) is another special folder that represents the parent folder of the one you are in.

What do you think happens at the root folder (/) if you cd ..? Have a look and find out.

If you re-run the last echo command with ls:

```
19 $ ls .*
```

you get a slightly more complicated output, as ls returns richer output depending on whether the item is a file or a folder. If it's a folder, it shows every (non-hidden) file within that folder under a separate heading.

Differences to 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.

```
21 $ rename -n 's/(.*)/new$1/' *
22 'file1' would be renamed to 'newfile1'
23 'file2' would be renamed to 'newfile2'
24 'file3' would be renamed to 'newfile3'
```

- Line 21 contains the command that renames all filenames to prepend 'new' in front. The `-n` flag tells rename to just print out the files that would be changed, and not actually carry out the renaming
- Lines 22-24 show the files that would be renamed

> **Note**
>
> You may not see the same output as above (or indeed any output), depending on your version of `rename`.

The first `*` character is treated as regular expressions, because it is not interpreted by the shell, but rather by the `rename` command. The reason it is not interpreted by the shell is because it is enclosed in single 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:

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

What You Learned

- What a glob is
- What a dotfile 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.

You will cover:

- Basic variables
- Quoting variables
- Quoting and globs
- The `env` and `export` commands
- Simple arrays

By the end you will have a good overview of how variables work in bash and some of their pitfalls.

How Important is this Section?

Variables are fundamental to understanding bash commands, much as they are in any programming language.

Basic Variables

Start by creating a variable and echoing it.

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

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

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

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

Variables and Quoting

Things get more interesting when you start quoting.

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

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

Since (by default) the shell reads each word in separated by a space, it thinks the word 'sentence' is not related to the variable assigment, and treats it as a program. To get the sentence into the variable with the space 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"
16 $ echo '$MYGLOB'
17 $ 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:

```
18 $ echo $PPID
19 $ PPID=nonsense
20 $ echo $PPID
```

- Line 18 - `PPID` is a special variable set by the bash shell. It contains the bash's parent process id.
- Line 19 - Try and set the `PPID` variable to something else.
- Line 20 - Output `PPID` again.

What happened there?

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

```
21  $ readonly MYVAR=astring
22  $ MYVAR=anotherstring
```

export

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

```
23  $ MYSTRING=astring
24  $ bash
25  $ echo $MYSTRING
26  $ exit
27  $ echo $MYSTRING
28  $ unset MYSTRING
29  $ echo $MYSTRING
30  $ export MYSTRING=anotherstring
31  $ bash
32  $ echo $MYSTRING
33  $ 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.

Outputting Exported and Shell Variables

Wherever you are, you can see the exported variables that are set by running `env`:

```
34 $ env
35 TERM_PROGRAM=Apple_Terminal
36 TERM=xterm-256color
37 SHELL=/bin/bash
38 HISTSIZE=1000000
39 TMPDIR=/var/folders/mt/mrfvc55j5mg73dxm9jd3n4680000gn/T/
40 PERL5LIB=/home/imiell/perl5/lib/perl5
41 GOBIN=/space/go/bin
42 Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.2BE31oXVrF/Render
43 TERM_PROGRAM_VERSION=361.1
44 PERL_MB_OPT=--install_base "/home/imiell/perl5"
45 TERM_SESSION_ID=07101F8B-1F4C-42F4-8EFF-1E8003E8A024
46 HISTFILESIZE=1000000
47 USER=imiell
48 SSH_AUTH_SOCK=/private/tmp/com.apple.launchd.uNwbe2XukJ/Listeners
49 __CF_USER_TEXT_ENCODING=0x1F5:0x0:0x0
50 PATH=/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/Users/imiell/google-clou\
51 d-sdk/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/space/git/shuti\
52 t:/space/git/work/bin:/space/git/home/bin:~/.dotfiles/bin:/space/go/bin
53 PWD=/space/git/work
54 LANG=en_GB.UTF-8
55 XPC_FLAGS=0x0
56 HISTCONTROL=ignoredups:ignorespace
57 XPC_SERVICE_NAME=0
58 HOME=/Users/imiell
59 SHLVL=2
60 PERL_LOCAL_LIB_ROOT=/home/imiell/perl5
61 LOGNAME=imiell
62 GOPATH=/space/go
63 DISPLAY=/private/tmp/com.apple.launchd.lwUJWwBy9y/org.macosforge.xquartz:0
64 SECURITYSESSIONID=186a7
65 PERL_MM_OPT=INSTALL_BASE=/home/imiell/perl5
66 HISTTIMEFORMAT=%d/%m/%y %T
67 HISTFILE=/home/imiell/.bash_history
68 _=/usr/bin/env
69 OLDPWD=/Users/imiell/Downloads
```

The output of `env` will likely be different wherever you run it.

That isn't all the variables that are set in your shell, though. It's just the *environment* variables that are exported to processes that you start in the shell.

If you want to see all the variables that are available to you in your shell, type:

```
69 $ compgen -v
```

`compgen` is a command that generates list of possible 'word completions' in bash when you hit tab repeatedly. The `-v` flag shows all the variables that could be completed in the context, which has the side effect here of listing all variables (exported and local to the shell) set where you are.

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:

```
70 $ bash --version
71 $ echo $BASH_VERSINFO
72 $ echo $BASH_VERSINFO[0]
73 $ echo ${BASH_VERSINFO[0]}
74 $ 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:

```
76 $ echo $BASH_VERSION_and_some_string
77 $ echo ${BASH_VERSION}_and_some_string
```

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

```
78 $ echo ${BASH_VERSION[0]}
```

So all bash variables are 'really' arrays!

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

```
79 $ echo ${BASH_VERSINFO[0]}
80 $ echo ${BASH_VERSINFO[1]}
81 $ echo ${BASH_VERSINFO[2]}
82 $ echo ${BASH_VERSINFO[3]}
83 $ echo ${BASH_VERSINFO[4]}
84 $ echo ${BASH_VERSINFO[5]}
85 $ echo ${BASH_VERSINFO[6]}
```

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

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 is 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. By the end you will have a more nuanced understanding of commands in bash.

How Important is this Section?

It's possible to get by without knowing much about functions in bash, but any serious bash user will know what they do and how they work.

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:

```
5  $ function myfunc {
6          echo $1
7          echo $2
8  }
9  $ myfunc "Hello World"
10 $ 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:

```
11 $ function myfunc {
12         echo $myvar
13 }
14 $ myfunc
15 $ myvar="Hi from outside the function"
16 $ 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:

```
17 $ function myfunc {
18         local myvar="Hi from inside the function"
19         echo $myvar
20 }
21 $ myfunc
22 $ echo $myvar
23 $ 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. It can't be declared outside a function.

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 four 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`?

```
25  $ function cd() {
26          echo 'No!'
27  }
28  $ cd /tmp
29  $ builtin cd /tmp
30  $ cd -
31  $ unset -f cd
32  $ cd /tmp
33  $ 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:

```
33  $ declare -f
34  $ 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.

```
35  $ which grep
36  $ which cd
37  $ which builtin
38  $ 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:

```
38 $ alias cd=doesnotexist
39 $ alias
40 $ cd
41 $ unalias cd
42 $ cd /tmp
43 $ cd -
44 $ alias
```

And yes, you can alias alias.

The 'type' Builtin

There is also a builtin called `type` that tells you how a command would be interpreted by the shell:

```
46 $ type ls
47 $ type pwd
48 $ type myfunc
```

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.

In this section you'll cover:

- Redirects
- Pipes
- File descriptors
- Special files like `/dev/null`
- Standard out (stdout) vs standard error (stderr)

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

How Important is this Section?

This section is essential reading.

Basic Redirects

Start off by creating a file:

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

The `>` character is the 'redirect' operator. This takes the output from the preceding command that you'd normally see in the terminal and sends it to a file that you give it. Here it creates a file called `file1` and puts the echoed string into it. Try `cat`ing the file if you want to check with: `cat file1`.

There's a subtlety here which we'll get to: sometimes not all the output you see in the terminal would get redirected by this, but don't worry about this yet.

Basic Pipes

Type this in:

```
4 $ 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 `file1` into the grep command by using the 'pipe' operator: `|`.

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

```
5 $ cat file2
```

What was the output of that?

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

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

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

> **Note**
>
> See Exercise 3 below for an often-useful operator that changes this behaviour.

Standard Output vs Standard Error

Before we get to 'standard output' and 'standard error', we need to take a step back.

In Linux (and UNIX), it's a fundamental design principle that 'everything is a file'. This means that when pushing data into things, or pulling data out of things, everything is treated as though it is a file.

For example, if you want to write to a terminal (or a 'simple' file, or a network interface), then you 'open' the file that corresponds to the terminal (or simple file, or network interface) and get back a 'file descriptor'.

> ## Note
>
> What is a 'simple file'?
>
> A 'simple' file here means a file that you would see in a folder. For example, a .txt file that just contains a note would be a 'simple' file.

The 'file descriptor' is a number associated with the process that represents that 'file'. You can then write to that 'file descriptor' and the operating system will take care of ensuring the data goes to the right place in the appropriate way.

Why did they design UNIX (and, later, the POSIX standard) this way? Because interacting with different types of data 'sinks' with a common interface means that you can stop worrying about how the data is managed behind the scenes, and just focus on what data goes in and comes out. For example, you don't care whether a simple file is a file

> ## Note
>
> What is a 'sink'?
>
> A 'sink' is an entity that you can push data into.

In the same way, you don't care if you are writing to a terminal or a file. Both have the same interface, so you just write your data to whichever file descriptor represents the thing you want to write to.

Why is this important for our use case here? In the case you saw above, the error when running `cat file2` was sent to a different file descriptor than the output when you ran `cat file1`. Remember, `file2` does not exist (so `cat'ing it throws an error), while` file1` does exist.

In UNIX/Linux each process gets three file descriptors by default. They are numbered zero to two:

- `0` is 'standard input'
- `1` is 'standard output'
- `2` is 'standard error'

When you ran `cat file1`, it took the filename you gave it and wrote the contents of that file out to the 'standard output' file descriptor `1`. By default, this is linked to the terminal, so you see this output on the terminal.

When you ran `cat file2`, it took the filename you gave it, and worked out that that filename did not exist. So it wrote the error message containing `No such file or directory` to the 'standard error' file descriptor `2`. By default, this is also linked to the terminal, so normally you would see no difference in the output compared to the 'standard output' of a 'normal' `cat` command.

So here is the key point: the `|` operator is a sink only for data passed into the 'standard output' file descriptor of the command on the left. Everything else it ignores. So when the command on the left (`cat` in this case) pushes data to the 'standard error' file descriptor, this still goes to the terminal.

Similarly, when you redirect output to a file, the `>` operator sends only the standard output to the file. Anything sent to standard error is not captured in the file.

What if you want to redirect standard error to a file? Try this:

```
7  $ command_does_not_exist
8  $ 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`.

When you use the standard redirection operator, you are implicitly using the file descriptor `1`. Therefore these two commands are equivalent:

```
9  $ echo "contents of file1" 1> file1
10 $ echo "contents of file1" > file1
```

The number before the `>` symbol indicates the file descriptor to redirect.

The file `/dev/null` is a special kind of sink 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 read in and ignored.

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

```
11 $ 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 (`&1`).

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 depending on the order of your operators. 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:

```
12 $ command_does_not_exist 2>&1 > outfile
13 $ 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!

Differences Between Pipes and Redirects

To recap:

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

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

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

The `<` operator redirects standard *input* (file descriptor `0`) *to* the command from a file, in this case having the same effect as `cat file1 | grep -c file` did.

```
15 $ echo line1 > file3
16 $ echo line2 > file3
17 $ echo line3 >> file3
18 $ 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:

```
19 $ cd ..
20 $ rm -rf lbthw_pipes_redirects
```

What You Learned

- What file redirection is
- 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'.

3) Research what the `|&` pipe operator does. If you can't find out, try the command `cat file2 |& grep -c file` and see what changes compared to when you ran it above.

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.

How Important is this Section?

This section is essential reading.

Shell Scripts

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

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

The Shebang

Run this:

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

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

Now try running it:

```
5 $ ./simple_script
```

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

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

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

This:

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

will give you the same output as before.

The Executable Flag

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

To correct this, run:

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

The chmod program changes the 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:

```
13 $ simple_script
```

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

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

If you run this you will see your `PATH` variable:

```
14 $ echo $PATH
```

Your output may vary. For example, mine is:

```
/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/Users/imiell/google-cloud-sdk\
/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:/space/git/shutit:/sp\
ace/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 sets up the `PATH` variable if you did not? The answer is: bash startup scripts.

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

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

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

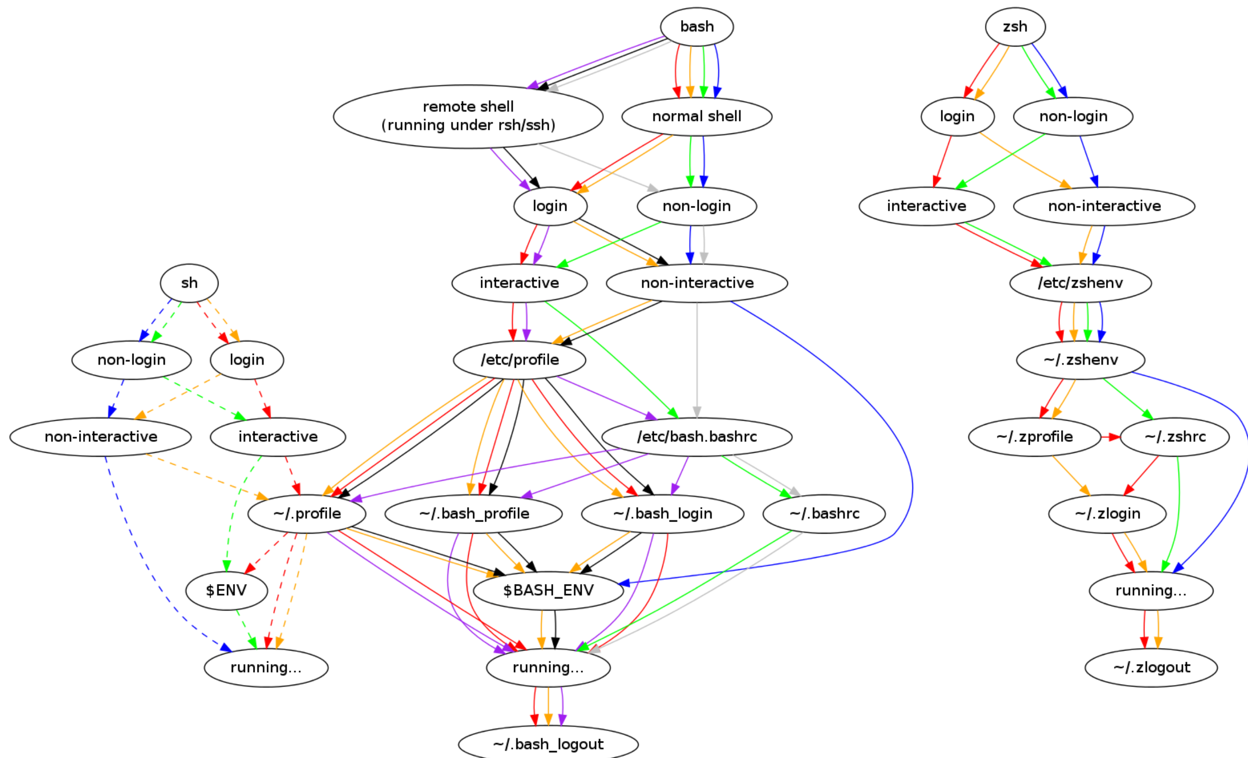
Startup Scripts

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 and 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 color 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 color 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 color 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!

The `source` Builtin

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

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

I'm sure you can figure out from that that `source` runs the script from within the same shell context. If you didn't see that, do it again, and 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.

Avoiding Startup Scripts

It's often useful to start up bash while avoiding all these startup scripts, to get bash in as raw a state as possible. When debugging, or testing core bash functionality, this can be invaluable.

To acheive that, run this:

```
22 $ env -i bash --noprofile --norc
```

`env` is a program that works on the environment. The effect of the `-i` flag is to remove the environment variables from the command you run. This means that exported variables will not get inherited by the bash program we are running. We're running the bash program itself with two flags. The `--noprofile` flag tells bash not to source the system-wide bash startup files, and the `--norc` tells bash not to source the personal ones in your home folder either.

The end effect of this is that your shell has a very minimal set of variables available:

```
23 $ env
24 $ exit
```

Cleanup

Now clean up what you just did:

```
25 $ cd ..
26 $ unset MYVAR
27 $ rm -rf lbthw_scripts_and_startups
```

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 your 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
- File substitution
- The internal field separator

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 substitution.

You will learn:

- What command substitution is
- The different ways of substituting commands

How Important is this Section?

The contents of this section are vital if you intend to write or work on shell scripts.

Command Substitution Example

An example may help illustrate. Type these commands:

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

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

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

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

Type this:

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

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 `echo` command takes the output of the above substitution, in this case nothing, as touch does not produce any output.

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 $ mkdir lbthw_cmdsub
12 $ cd lbthw_cmdsub
13 $ echo `touch \`ls ..\``
14 $ 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.

To demonstrate the difficulty here, a simple example suffices:

```
15 $ echo $(echo hello1 $(echo hello2))
16 $ echo `echo hello1 `echo hello2``
17 $ echo `echo hello1 \`echo hello2\``
```

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:

```
18 $ echo `echo \`echo \\\`echo inside\\\`\``
19 $ echo $(echo $(echo $(echo inside)))
```

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

Cleanup

Remove the left-over directory:

```
20 $ rm -rf lbthw_cmdsub
```

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

Exercises

1) Try various command substitution commands, and plug in variables and quotes to see what happens.

2) Explain why three backslashes are required in the last example.

Exit Codes

Before you cover tests and special parameters in bash, a crucial and related concept to grasp is exit codes.

You will cover:

- What an exit code is
- How to set one in a script and a function
- Exit code conventions
- Some other 'special' parameters in bash

How Important is this Section?

Most bash scripts won't be completely comprehensible or safely written unless exit codes are understood.

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:

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

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

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

```
5  $ bash
6  $ function trycmd {
7          $1
8          if [[ $? -eq 127 ]]
9          then
10                 echo 'What are you doing?'
11         fi
12 }
13 $ trycmd ls
14 $ trycmd doesnotexist
15 $ exit
```

Note

There is an `if` statement above. We will cover `if` statements (and tests in the next section).

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 of a non-standard program, 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	Problem when running builtin command
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

Number	Meaning	Notes
		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:

```
16 $ echo 'grepme' > afile.txt
17 $ grep not_there afile.txt
18 $ echo $?
```

Did you expect that? grep finished successfully (there was no segmentation or memory fault, 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:

```
19 $ if grep grepme afile.txt
20 then
21         echo 'matched!'
22 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, you can set the exit code of the function by using the `exit` builtin. You can simulate this simply by entering a new bash process and then exiting from it.

```
23 $ bash
24 $ exit 67
25 $ echo $?
```

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

Type this:

```
26 $ bash
27 $ function trycmd {
28         $1
29         if [[ $? -eq 127 ]]
30         then
31                 echo 'What are you doing?'
32                 return 1
33         fi
34 }
35 $ trycmd ls
36 $ trycmd doesnotexit
37 $ 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:

```
38 $ ps -ef | grep bash | grep $$
39 $ sleep 999 &
40 $ 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:

```
41 $ 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 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) 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.

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.

A test in bash is not like a test that your program works. It's a way of writing an expression that can be true or false.

In this section you will learn about:

- What bash tests are
- Different ways of writing bash tests
- Logical operators
- Binary and unary operators
- `if` statements

How Important is this Section?

This section is very important, as it covers material essential to if or while statements in bash as well as writing more complex chains of commands.

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:

```
1  $ [ 1 = 0 ]
2  $ echo $?
3  $ [ 1 = 1 ]
4  $ 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 result of the last-executed command. If it returned true, the number will (usually) be '0'. If it didn't, the number will (usually) *not* be '0'.

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

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

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

What is '[', Really?

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

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

and that `test` and `[ ... ]` are more or less synonymous:

```
14  $ which test
15  $ man test # Hit q to get out of the manual page.
16  $ man [ # Takes you to the same page.
```

> **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.

Logical Operators

What do you expect the output of this to be?

```
17 $ ( [ 1 = 1 ] || [ ! '0' = '0' ] ) && [ '2' = '2' ]
18 $ 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.

From here on in in this book I will use logical operators (particularly `&&`) where it's appropriate to. This should get you used to the idiom.

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

```
19 $ [  1 = 1 -o  ! '0' = '0'  -a  '2' = '2' ]
20 $ 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. I have not seen these used often in the field. You can mostly ignore 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:

```
21 $ [[ 1 = 1 ]]
22 $ 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:

```
23 $ unset DOESNOTEXIST
24 $ [ ${DOESNOTEXIST} = '' ]
25 $ echo $?
26 $ [[ ${DOESNOTEXIST} = '' ]]
27 $ echo $?
28 $ [ x${DOESNOTEXIST} = x ]
29 $ 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:

```
[ = '' ]
```

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:

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

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:

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

You can frequently come across code like this:

```
30 $ [[ "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. The first set take a single argument and are called 'unary' operators:

```
31 $ echo $PWD
32 $ [ -z "$PWD" ]
33 $ echo $?
34 $ unset DOESNOTEXIST
35 $ [ -z "$DOESNOTEXIST" ]
36 $ echo $?
37 $ [ -z ]
38 $ 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`:

```
39 $ mkdir lbthw_tests_dir
40 $ touch lbthw_tests_file
41 $ [ -a lbthw_tests_file ]
42 $ echo $?
43 $ [ -d lbthw_tests_file ]
44 $ echo $?
45 $ [ -a lbthw_tests_dir ]
46 $ echo $?
47 $ [ -d lbthw_tests_dir ]
48 $ echo $?
49 $ rmdir lbthw_tests_dir
50 $ rmdir lbthw_tests_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:

```
69  $ if [[ 10 -lt 2 ]]
70  then
71    echo 'does not compute'
72  elif [[ 10 -gt 2 ]]
73  then
74    echo 'computes'
75  else
76    echo 'does not compute'
77  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:

```
78  $ 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!

```
79  $ if grep not_there /dev/null
80  then
81      echo there
82  else
83      echo not there
84  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.

You will cover:

- 'C'-style 'for' loops
- 'for' loops over items 'in' a list
- 'while' and 'until' loops
- 'case' statements

How Important is this Section?

'For' loops are a basic construct in most languages, and bash is no exception.

`for` Loops

Create and move into a new folder:

```
1 $ mkdir lbthw_loops && cd lbthw_loops
```

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

```
2 $ for (( i=0; i < 20; i++ ))
3 do
4   echo $i
5   echo $i > file${i}.txt
6 done
7 $ 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!

```
7   $ for f in $(ls *txt)
8   do
9     echo "File $f contains: $(cat $f)"
10  done
```

It's our old friend the command substitution! The command substitution 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:

```
11  $ for f in file1.txt file2.txt file3.txt
12  do
13    echo "File $f contains: $(cat $f)"
14  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:

```
15  $ n=0
16  $ while [[ ! -a newfile ]]
17  do
18          ((n++))
19          echo "In iteration $n"
20          if [[ $(cat file${n}.txt) == "15" ]]
21          then
22                  touch newfile
23          fi
24  done
```

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

```
25  $ n=0
26  $ while true
27  do
28          ((n++))
29          echo $n seconds have passed
30          sleep 1
31          if [[ $n -eq 60 ]]
32          then
33                  break
34          fi
35  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:

```
36  $ a=1
37  $ case "$a" in
38  1) echo 'a is 1'; echo 'ok';;
39  2) echo 'a is 2'; echo 'ok';;
40  *) echo 'a is unmatched'; echo 'failure';;
41  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 a file (case.sh) to try out a case statement with getopts:

```
42 $ cat > case.sh << 'EOF'
43 #!/bin/bash
44 while getopts "ab:c" opt
45 do
46     case "$opt" in
47     a) echo '-a invoked';;
48     b) echo "-b invoked with argument: ${OPTARG}";;
49     c) echo '-c invoked';;
50     esac
51 done
52 EOF
53 $ chmod +x case.sh
```

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

```
54 $ ./case.sh -a
55 $ ./case.sh -b
56 $ ./case.sh -b "an argument"
57 $ ./case.sh -a -b -c
58 $ ./case.sh
```

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

Cleanup

```
59 $ 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 bash options, and the `set` builtin.

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.

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.

You will cover:

- The `set` command
- What POSIX is
- Some useful options to set when scripting
- What the `pipefail` option is used for

How Important is this Section?

Setting options is not absolutely core material, but once you've used bash for a while, understanding them becomes important. It's also good revision material for some of the concepts covered previously regarding exported vs non-exported variables.

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:

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

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

> **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:

```
6  $ set -o
```

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

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

`set` VS `env`

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

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

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

Useful Options for Scripting

Where set becomes really useful to understand is in scripting.

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

```
9  $ set -o errexit
10 $ set -o xtrace
11 $ set -o nounset
```

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 being run.

The `nounset` option gets bash to throw an error if a variable is not set when it is referred to.

Type this is to see how this works in practice:

```
12 $ echo '#!/bin/bash
13 set -o errexit
14 set -o xtrace
15 set -o nounset
16 pwd
17 cd $HOME
18 cd -
19 echo $DOESNOTEXIST
20 echo "should not get here"' > ascript.sh
21 $ chmod +x ascript.sh
22 $ ./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:

```
23 $ set -e
24 $ set -x
```

is the same as:

```
25 $ set -o errexit
26 $ set -o xtrace
```

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

The `pipefail` Option, Exit Codes and Pipelines

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

```
27 $ touch afile.txt
28 $ set -o pipefail
29 $ grep notthere afile.txt | xargs
30 $ echo $?
```

As you know, the grep that doesn't find anything to match returns an exit code of 1 (false). This output is passed to xargs, which always returns a zero (true). So the question here is what should be returned? The answer depends on the pipefail setting.

```
31 $ set +o pipefail
32 $ grep notthere afile.txt | xargs
33 $ echo $?
```

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.

`set` VS `shopt`

Although we don't cover it in depth in this section, it's worth mentioning that there are two ways to set bash options from within scripts or on the command line. You can use the `set` builtin command, or the `shopt` builtin command. They both manipulate the behaviour of the shell, and differ for historical reasons. The set options are inherited, or borrowed, from other shells' options, while the `shopt` ones (mostly) originated in bash.

Just to demonstrate one option that you might find useful, the `globstar` option allows you to use two asterisks to match all files in the local directory and all subdirectories:

```
34 $ shopt -s globstar
35 $ ls **
```

If you have a lot of files in your subfolders, then it might take a long time to return!

Cleanup

Now clean up what you just did:

```
36 $ 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?

Next we cover process substitution, a tricky but useful concept to master for sophisticated bash scripting.

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.

Process Substitution

Process substitution is a really handy way of saving time at the command line.

In this section you will cover:

- The `<()` operator
- The more rarely-used `>()` operator

How Important is this Section?

I spent years reading and writing bash before I understood this concept, so it's possible to skip this section. However, once I learned it, I use it on the command line almost every day, so I recommend you learn it at some point.

Simple Process Substitution

Type this in to set files up for this section:

```
1  $ mkdir lbthw_process_subst && cd lbthw_process_subst
2  $ mkdir a
3  $ mkdir b
4  $ touch a/1 a/2
5  $ touch b/2 b/3
6  $ ls a
7  $ ls b
```

You've created two folders with slightly different contents.

Now let's say that you want to diff the output of `ls a` and `ls b` (a trivial but usefully simple example here). How would you do it?

You might do it like this:

```
8  $ ls a > aout
9  $ ls b > bout
10 $ diff aout bout
11 $ rm aout bout
```

That works, and there's nothing wrong with it, but typing all that out and cleaning up the files is a bit cumbersome. There's a much neater way that exposes a very useful technique.

Type this in:

```
12 $ diff <(ls a) <(ls b)
```

That's neater, isn't it?

So what's going on?

The `<()` Operator

The `<()` operator is conceptually similar to the `$()` we saw ealier. In the same way that `$()` substitutes the *output* of the process contained within it into the command, eg:

```
13 $ echo $(ls a)
```

the `<()` operator substitutes a *file containing the output* of the process contained within it. You might need to stop and think about this for a second.

That means that this line:

`diff <(ls a) <(ls b)`

effectively becomes the command to diff two files, equivalent to the files `aout` and `bout` in the lines you typed in above.

So *wherever you would normally put a filename*, you can use the `<()` operator to save some time by dropping these in rather than creating files.

The `>()` Operator

Can you guess what this does? It's similar to the `<()` but for me it was a lot trickier to grasp, and much more rarely seen (so feel free to skip).

See if you can work out from this line what it does:

```
14 $ tar cvf >(cat > out.tar) /tmp
```

As with the `<()` operator, this replaces a file in a command. This time, rather than sending the output to the file, it takes *input* from the command that would normally go to that file reference, and feeds that input to the command in the operator.

Let's take a step back and think about that, because it can be hard to follow.

Normally you'd write something like this:

```
15 $ rm -f out.tar
16 $ tar cvf out.tar /tmp
```

The command is read into bash, expanded out, and the tar command accepts two arguments: a file and a folder (`out.tar` and the `/tmp`). It tars up the contents of the `/tmp` folder and places it in the `out.tar` file.

The difference in the previous command is that the contents that would normally be inputted into the file is instead fed into the command `cat > out.tar`.

Obviously, in this case that command is pointless - in both cases you end up with a file called `out.tar` that is a tar file.

Let's say, however, that you wanted to use a different compression scheme for your tar file. You could type this:

```
17 $ tar cvf >(gzip > out.tar.gz) /tmp
```

which would gzip the tarfile and place it into the `out.tar.gz` file.

It can reasonably be pointed out that most versions of tar offer a gzip flag (`-z`). that does this for you. However, some versions don't (especially on minimal Linux distributions like busybox), so this can be a neat way of getting round that.

I have never had a need to use this mechanism in real life, but I've written things like this before, which are less neat (but good enough):

```
18 $ tar cvf out.tar /tmp
19 $ gzip out.tar
20 $ rm out.tar
```

Cleanup

To clean up:

```
21 $ cd .. && rm -rf lbthw_process_subst
```

What You Learned

- What the `<()` operator is
- What the `>()` operator is
- How to use these operators
- How `<()` differs from the `$()` operator
- How `>()` works

What Next?

Next you will cover subshells, and grouping commands more generally.

Exercises

1) Look for examples of where these operators are used on the web, and figure out what they're doing.

2) Look through your bash history (by typing `history`) and see where you could have used these operators.

3) Construct a command that uses `$()`, `<()`, and `>()`.

Subshells

The concept of subshells is not a complicated one, but can lead to a little confusion at times, and occasionally is very useful.

You will cover:

- How to create a subshell
- Why they are useful
- Comparison to curly-braced blocks

How Important is this Section?

As with process substitution, this is another concept I came to later in my bash career. It comes in handy fairly often, and an understanding of it helps deepen your bash wisdom.

Create a Subshell

Type this in to see a subshell in action:

```
1 $ mkdir lbthw_subshells && cd lbthw_subshells
```

Create a variable in your 'main' shell:

```
2 $ VAR1='the original variable'
```

Now create a 'subshell':

```
3 $ (
```

You'll notice the prompt has changed. Now try and echo something:

```
4 > echo Inside the subshell
```

It's not been run. This is because the subshell's instructions aren't run until the parenthesis has been closed. Next we'll try and echo the variable we created outside.

```
5 > echo ${VAR1}
```

Then update that variable to another value, and echo it again:

```
6  > VAR1='the updated variable'
7  > echo ${VAR1}
```

And finally create another variable, before closing the subshell out:

```
8  > VAR2='the second variable'
9  > )
```

So a subshell is a shell that inherits variables from the parent shell, and whose running is deferred until the parentheses are closed.

It will pay to think about the output and review the subshell commands to grasp what you just saw. Play with the commands and experiment until you're comfortable with what a subshell can and can't do.

Subshells and Scope

What happened to the variable you updated inside the subshell?

```
10 $ echo ${VAR1}
```

Now you now know that variables can mask their parent shell's value within the subshell. This is very handy to know if you want to do something in a slightly modified environment, but not have to manage variables' previous values.

What happens if we try `export`ing the variable? Will that 'export' it to the parent shell?

```
11 $ (
12 > export VAR1='the first variable exported'
13 > )
14 $ echo ${VAR1}
```

The `export` Builtin

The `export` command can cause a lot of confusion here. Let's experiment with it here to show how it works.

> **Note**
>
> The next code snippet puts quotes around the `END` delimiter. We will cover why in the next part of the book.

```
15  $ cat > echoes.sh << 'END'
16  > #!/bin/bash
17  > echo $EXPORTED
18  > echo $NOTEXPORTED
19  > 'END'
```

Then run this. What do you think will happen?

```
20  $ chmod +x echoes.sh
21  $ export EXPORTED='exported'
22  $ NOTEXPORTED='exported'
23  $ ./echoes.sh
```

If this doesn't make sense to you, you might want to revisit Part I, where `export` was covered in 'Variables'.

Subshells and Redirection

One often useful property of subshells is the fact that the code is treated as a single unit. You can therefore redirect output from a set of commands wholesale.

You might write code that looks like this

```
24  $ echo Some output, ls to follow >> logfile
25  $ ls >> logfile
```

which is fine for a couple of lines, but what if you have hundreds of lines like this? It leads to quite inelegant code. Instead, you can write:

```
26  $ (
27  > echo Some output, ls to follow
28  > ls
29  > ) >> logfile
```

which is much neater and easier to manage.

`()` vs `{}`

One source of confusion is the difference between these two ways of grouping commands.

First, try replacing the `(` and `)` in the above listing with curlies (`{` and `}`) and see what happens.

Then, try these:

```
30 $ ( echo output ) >> logfile
31 $ { echo output } >> logfile
```

Hmmm. Hit `CTRL+c` to get out.

The curlies need a semicolon to indicate the end.

```
32 $ { echo output ; } >> logfile
```

This is because the curlies are a *grouping command* and need an indication of when the command has been completed. No subshell is created. The environment, variables, current working directory, indeed the entire context are the same as the surrounding code.

Subshells and Working Directory

Another useful feature of subshells is that if you change folder, then that folder change only applies within the subshell. When you return, you go back to where you were.

Let's see this in action. First show where you are:

```
33 $ pwd
```

Then, in a subshell, create a folder, move into it, and show you have moved with another call to pwd.

```
34 $ ( mkdir lbthw_subshells_2
35 > cd lbthw_subshells_2
36 > pwd )
```

Now the subshell has run, and without doing anything other than `cd`ing to a new folder, we are returned to the folder we were in before the subshell started:

```
35 $ pwd
```

This is a very useful feature of subshells when you're scripting, to save repeated `cd` and `cd -` commands.

Cleanup

To clean up:

```
36 $ cd .. && rm -rf lbthw_subshells
```

What You Learned

- How to create a subshell
- The difference between `()` and `{}`
- How variable scoping works in subshells and grouping commands
- Redirection in subshells and groups

What Next?

Next you will look at a common challenge when dealing with files and for loops: the internal field separator.

Exercises

1) Show that the group command (`{` and `}`) does affects the outer shell's working directory, while a subshell does.

2) Work out what the `BASH_SUBSHELL` variable does.

Internal Field Separator

This mouthful is important to know about if you are going to shell script.

You will cover:

- What the `IFS` variable is for
- When it should be used
- How to set it up
- Ways of dealing with odd filenames in loops

How Important is this Section?

Learning this concept will save you a great deal of time trying to figure out why your for loop is not working as expected, and will help you write more correct bash scripts that won't fail.

Files With Spaces

Create a folder to work in:

```
1 $ mkdir lbthw_ifs && cd lbthw_ifs
```

Now create a couple of files with spaces in:

```
2 $ echo file1 created > "Spaces in filename1.txt"
3 $ cat "Spaces in filename1.txt"
4 $ echo file2 created > "Spaces in filename2.txt"
5 $ cat "Spaces in filename2.txt"
```

Note that you have to quote the filename to get the spaces in. Without the quotes, bash treats the space as a token separator. This means that it would treat the redirection as going to the file 'Spaces' and not know what to do with the 'in' and 'filenameN.txt' tokens.

Now if you write a for loop over these files that just runs `ls` on each file in turn, you might not get what you expect. Type this in and see what happens:

```
6 $ for f in $(ls)
7 > do
8 >   ls $f
9 > done
```

Hmmm. The for loop has treated every word in the filenames as a separate token to look for with `ls`.

In other words, bash has treated each space as a 'field separator'. Normally this is fine, as our for loops have items separated by spaces, like this:

```
10  $ for f in 1 2 3 4
11  > do
12  >   echo $f
13  > done
```

However, here we want the spaces to be ignored. And we can control this by setting the `IFS` shell variable.

The IFS Shell Variable

```
14  $ echo $IFS
```

If you retained the default, then you will have seen nothing in the output, which isn't very helpful. To see how it's really set up, we can use `set`:

```
15  $ set | grep IFS
```

You should see output like this:

```
IFS=$' \t\n'
```

Recall that the `$` before the single quotes means that the variable is showing you special characters with the backlash escape notation. Above, the `IFS` variable is set to: space, tab, and newline.

Bash takes the characters in that variable and will treat any of them as a field separator, which means that the original for loop you wrote above will create from these files:

```
Spaces in filename1.txt
Spaces in filename2.txt
```

these list of items:

```
Spaces
in
filename1.txt
Spaces
in
filename2.txt
```

What we want to do is treat the spaces like any other character. We can do this by altering the IFS variable to remove the space and re-running the for loop:

```
16 $ IFS=$'\t\n'
17 $ for f in $(ls)
18 > do
19 >   ls $f
20 > done
```

This might sound obscure but it's quite frequent in bash to perform operations over bunches of files like this:

```
$ find . -type f | xargs -n1 grep somestring
```

and in these cases you may have files lying around that have spaces in their names. Then your scripts can fail in ways that it can be difficult to debug when you don't know about the `IFS` variable.

> ## Note
>
> The `find` program (as the name suggests) helps you find files. If you just give it a folder (as above with the '.' local directory) then it will return all files and folders under that folder. The `-type f` flag tells find to just return files, not folders. The `xargs` command runs the command given against the files piped in, and the `-n1` flag tells xargs to run the command once per field piped in.

The Null Byte as Separator

While we're on the subject, it's worth mentioning quickly a pattern that's very commonly-used to get around the above scenario.

```
$ find . -type f -print0 | xargs -0 -n1 grep somestring
```

By adding the `-print0` flag, find no longer uses a new line as a field separator. It uses what's called a `NUL` byte as the separator. The `NUL` byte is literally a byte of value zero. It doesn't get displayed on the screen, but can be read by `xargs` as the separator if it's given the `-0` flag. This bypasses all sorts of challenges you might get fiddling with the `IFS` variable or dealing with spaces or other odd characters in filenames.

Cleanup

To clean up:

```
21 $ cd .. && rm -rf lbthw_ifs
```

What You Learned

- What the `IFS` shell variable does
- How to deal with filenames containing spaces and other unusual characters
- What the `find` and `xargs` programs do
- What a `NUL` byte is

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) Set the `IFS` variable to various characters and demonstrate how field separation works

2) Run the output of `find . -type f` through hexdump and pick out the `NUL` bytes in the stream

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:

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

Readline

Readline is one of those technologies that is so commonly used people don't realise it's there. In this section I want to make you aware of it and introduce some concepts and examples of its use so you're able to deal with any problems that arise from its use or misuse.

In this section you'll learn about:

- What readline is
- What bash looks like without it
- How readline is commonly used
- How shortcuts get interpreted by the system

How Important is this Section?

Writing bash does not require a knowledge of readline. Understanding readline helps greatly to understand what is going on at the terminal and at the command line.

Bash Without Readline

First you're going to see what bash looks like without readline.

In your 'normal' bash shell, hit the 'tab' key twice. You should see something like this:

```
Display all 2335 possibilities? (y or n)
```

That's because bash normally has an 'autocomplete' function that allows you to see what commands are available to you if you tap tab twice.

Hit 'n' to get out of that autocomplete.

Similarly, if you hit the up arrow key a few times, then the previously-run commands should be brought back to the command line.

Now type:

```
1 $ bash --noediting
```

The `--noediting` flag starts up bash without the readline library enabled. If you hit tab twice now you will see something different: the shell no longer 'sees' your tab and just sends a tab direct to the screen. Autocomplete has gone.

Autocomplete is just one of the things that the readline library give you in the terminal. You might want to try hitting the up or down arrows as you did above to see that that no longer works as well.

Hit return to get a fresh command line, and exit your non-readline-enabled bash shell:

```
2 $ exit
```

Other Shortcuts

There are a great many shortcuts like autocomplete available to you if readline is enabled. I'll quickly outline four of the most commonly-used of these before explaining how you can find out more.

```
3 $ echo 'some command'
```

There should not be many surprises there. Now if you hit the 'up' arrow, you will see you can get the last command back on your line. If you like, you can re-run the command, but there are other things you can do with readline before you hit return.

If you hold down the 'ctrl' key and then hit 'a' at the same time your cursor will return to the start of the line. Another way of representing this 'multi-key' way of inputting is to write it like this: '`\C-a`'. This is one conventional way to represent this kind of input. The '`\C`' represents the control key, and the '`-a`' represents that the 'a' key is depressed at the same time.

Now if you hit '`\C-e`' ('ctrl' and 'e') then your cursor has moved to the end of the line. I use these two dozens of times a day.

Another frequently useful one is '`\C-l`', which clears the screen, but leaves your command line intact.

The last one I'll show you allows you to search your history to find matching commands while you type. Hit '`\C-r`', and then type '`ec`'. You should see the `echo` command you just ran like this:

```
(reverse-i-search)`ec': echo echo
```

Then do it again, but keep hitting '`\C-r`' over and over. You should see all the commands that have `ec` in them that you've input before (if you've only got one echo command in your history then you will only see one). As you see them you are placed at that point in your history and you can move up and down from there or just hit return to re-run if you want.

> What is 'history'?
>
> History is a list of commands that were previously-ran in your shell. We will cover this in a later section.

There are many more shortcuts that you can use that readline gives you. Next I'll show you how to view these.

Using `bind` to Show Readline Shortcuts

If you type:

```
4 $ bind -p
```

You will see a list of bindings that readline is capable of. There's a lot of them! Have a read through if you're interested, but don't worry about understanding them all yet.

If you type:

```
5 $ bind -p | grep C-a
```

you'll pick out the 'beginning-of-line' binding you used before, and see the '`\C-a`' notation I showed you before.

As an exercise at this point, you might want to look for the '`\C-e`' and '`\C-r`' bindings we used previously.

If you want to look through the entirety of the `bind -p` output, then you will want to know that `\M` refers to the `Meta` key (which you might also know as the `Alt` key), and `\e` refers to the `Esc` key on your keyboard. The 'escape' key bindings are different in that you don't hit it and another key at the same time, rather you hit it, and then hit another key afterwards. So, for example, typing the `Esc` key, and then the '`?`' key also tries to auto-complete the command you are typing. This is documented as:

```
"\e?": possible-completions
```

in the `bind -p` output.

Readline and Terminal Options

If you've looked over the possibilities that readline offers you, you might have seen the '`\C-r`' binding we looked at earlier:

```
"\C-r": reverse-search-history
```

You might also have seen that there is another binding that allows you to search forward through your history too:

```
"\C-s": forward-search-history
```

What often happens to me is that I hit '`\C-r`' over and over again, and then go too fast through the history and fly past the command I was looking for. In these cases I might try to hit '`\C-s`' to search forward and get to the one I missed.

Watch out though! Hitting '`\C-s`' to search forward through the history might well not work for you.

Why is this, if the binding is there and readline is switched on?

It's because something picked up the '`\C-s`' before it got to the readline library: the terminal settings.

The terminal program you are running in may have standard settings that do other things on hitting some of these shortcuts before readline gets to see it.

If you type:

```
5 $ stty -e
```

you should get output similar to this:

> **Note**
>
> If you get an error running `stty -e` (here or later) then try `stty -a` instead, which gives effectively the same output, but is slightly harder to read on the terminal.

```
speed 9600 baud; 47 rows; 202 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -iutf8
        -ignbrk brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts -dsrflow
        -dtrflow -mdmbuf
discard dsusp   eof     eol     eol2    erase   intr    kill    lnext
^O      ^Y      ^D      <undef> <undef> ^?      ^C      ^U      ^V
min     quit    reprint start   status  stop    susp    time    werase
1       ^\      ^R      ^Q      ^T      ^S      ^Z      0       ^W
```

You can see on the last four lines there is a table of key bindings that your terminal will pick up before readline sees them. The '`^`' character (known as the 'caret') here represents the '`ctrl`' key that we previously represented with a '`\C`'.

If you think this is confusing I won't disagree. Unfortunately in the history of Unix and Linux documenters did not stick to one way of describing these keys.

If you encounter a problem where the terminal options seem to catch a shortcut key binding before it gets to readline, then you can use the `stty` program to unset that binding. In this case, we want to unset the 'stop' binding.

If you are in the same situation, type:

```
6 $ stty stop undef
```

Now, if you re-run `stty -e`, the last two lines might look like this:

```
[...]
min     quit    reprint start   status  stop    susp    time    werase
1       ^\      ^R      ^Q      ^T      <undef> ^Z      0       ^W
```

where the `stop` entry now has '`<undef>` underneath it.

Strangely, for me '`C-r`' is also bound to 'reprint' above (`^R`). But (on my terminals at least) that gets to readline without issue as I search up the history. Why this is the case I haven't been able to figure out. I suspect that reprint is ignored by modern terminals that don't need to 'reprint' the current line.

While we are looking at this table:

```
discard dsusp   eof     eol     eol2    erase   intr    kill    lnext
^O      ^Y      ^D      <undef> <undef> ^?      ^C      ^U      ^V
min     quit    reprint start   status  stop    susp    time    werase
1       ^\      ^R      ^Q      ^T      <undef> ^Z      0       ^W
```

it's worth noting a few other key bindings that are used regularly.

First, one you may well already be familiar with is '`\C-c`', which interrupts a program, terminating it:

```
7  $ sleep 99
8  [[Hit \C-c]]
9  ^C
10 $
```

Similarly, '`\C-z`' suspends a program, allowing you to 'foreground' it again and continue with the `fg` builtin.

```
10 $ sleep 10
11 [[ Hit \C-z]]
12 ^Z
13 [1]+  Stopped                 sleep 10
14 $ fg
15 sleep 10
```

The '`\C-d`' sends an 'end of file' character. It's often used to indicate to a program that input is over. If you type it on a bash shell, the bash shell you are in will close.

Finally, '`\C-w`' deletes the word before the cursor

These are the most commonly-used shortcuts that are picked up by the terminal before they get to the readline library.

What You Learned

- What the readline library is
- What features it gives you
- How to find out more about those features
- The order in which terminal options and readline are considered
- Some terminal-native shortcuts
- How key combinations are represented in standard documentation

What Next?

Armed with your increased knowledge of how the readline library works, and terminal program settings, you are ready to tackle the tricky subject of terminal codes.

Cleanup

You may want to close your bash shell (perhaps with '`\C-d`'!) and open up a fresh one as you might have tinkered with your default settings above.

Exercises

1) Research the terminal options' shortcuts in the `stty -e` table and what they mean.

2) Research the inputrc file and edit it to configure how `backward-kill-word` is invoked.

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.

You've already learned about the readline library and terminal options and how certain keystrokes can be 'caught' and handled in other ways before they get to the terminal. Here we look at how other non-standard can be handled in the bash shell.

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 if they get that far.

In this section you'll:

- Understand how to output any byte with echo
- Understand how to input a character while by-passing terminal options and the readline library
- 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 escape codes. Rarely, you might come across more complex escapes for specific terminal contexts, but this is beyond the scope of a 'practical' guide.

How Important is this Section?

This section is somewhat more advanced, and probably not essential to using bash at the start. It can also be hard to grasp, so if you're new to bash I recommend returning to it later if it's hard to understand the first time you read it.

However, knowledge of this area will catapult you to an elite that understand how terminals can be manipulated, and also enable you to understand how your prompt can be manipulated.

Non-Printable Characters

The terminal you use has what are described can output '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 necessarily involve writing a character you'd recognise.

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 color 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 non-printable 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 you've not already used in this book:

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

The -n flag tells echo not to print a newline.

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

The `$` before the string makes us sure that bash will interpret `\n` as a newline on the screen, and won't just print out the characters`\n`. Try it without the `$` if you're unsure.

Here, it is the backslash character \ that makes `echo` aware that a 'special' character is coming up. Being able to add a newline in this way means that you can send a newline to the terminal via the echo command, without confusing the command line by hitting the return key.

Other special characters include `\b` (for backspace), `\t` (for tab) and `\\` (to output a backslash):

Before you hit return, have a guess as to what this will output?

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

If you guessed correctly, then well done! If you're struggling to understand what happened, note that a backspace does not delete the previous character, it just moves the cursor back a space.

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

```
4 $ echo -n $'\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, or controlling what gets sent to the terminal.

It also bypasses the 'catching' of some characters you've seen from the previous readline section.

`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 bypass the terminal's interpretation via terminal options or the readline library?

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 readline library uses the tab key (if hit twice in a row) to auto-complete any text we have not finished.

But if I'm typing something like:

```
5 $ echo 'I want a tab here:>X<a tab'
```

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

This is one way: instead of the 'X', you type '`\C-v`' and then '`\C-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.

If you look at the output of `stty -e` again (as you did in the previous readline section) then you will that `^V` is bound to the 'lnext' action. I believe (but can't confirm) that this stands for 'next character literal'.

```
discard dsusp   eof     eol     eol2    erase   intr    kill    lnext
^O      ^Y      ^D      <undef> <undef> ^?      ^C      ^U      ^V
min     quit    reprint start   status  stop    susp    time    werase
1       ^\      ^R      ^Q      ^T      ^S      ^Z      0       ^W
```

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
CR	0x0D	\r	^M	Carriage return
ESC	0x1B	N/A	^[	Escape character

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

```
6 $ echo abcc^Hdefg
```

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 strange 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.

So what will this output?

```
7 $ echo $'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:

```
8 $ echo $'Bad magazine\rMad' | hexdump
9 $ echo $'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

A terminal escape code is a defined sequence of bytes that, if sent to the terminal, will perform a specific action.

Run this:

```
$ echo $'\033[?47h'
$ echo $'\033[?47l'
```

The first line 'saves' the screen and the second restores it.

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

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

These ESC and left bracket characters are then followed by specific sequences which can change the color 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}
}
```

```
54
55 trap 'exit 1' INT TERM
56 trap 'tput setaf 9; tput cvvis; clear' EXIT
57
58 tput civis
59 clear
60
61 while :; do
62     for ((c=1; c <= 7; c++)); do
63         tput setaf $c
64         for ((x=0; x<${#DATA[0]}; x++)); do
65         for ((y=0; y<=4; y++)); do
66         draw_char $x $y
67         done
68         done
69     done
70 done
```

What You Learned

- What terminal codes are
- What printable and non-printable characters are
- How to output any arbitrary bytes
- How to input a literal 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.

You will learn:

- How to set up your prompt
- Other types of prompts and related shell variables
- How to put more useful info in your prompt

How Important is this Section?

This section is not essential, but most people find it interesting and maybe fun to learn about.

The PS1 Variable

Type this:

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

```
5  $ bash
6  $ PS2='I am in the middle of something!>>>  '
7  $ cat > /dev/null << END
8  some text
9  END
10 $ 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 book as I've barely ever seen it used.

PS4 is the last one:

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

```
19  $ bash
20  $ PS1='\u@\H:\w \# \$ '
21  $ ls
22  $ 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:

```
23  $ bash
24  $ PS1='\[\033[01;31m\]PRODUCTION\$ '
25  $ PS1='\[\033[01;32m\]DEV\$ '
26  $ 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`:

```
27  $ bash
28  $ PROMPT_COMMAND='echo "Hello prompt $(date)"'
29  $ ls
30  $ 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 the content of files directly in your scripts without too much overhead.

> **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.

You will cover:

- What 'here documents' are
- What 'here strings' are
- How they are used in shell scripts

How Important is this Section?

Here documents, once learned, are used frequently in bash scripts or on the command line. It's a piece of knowledge that separates the experienced user from the junior.

Basic Here Docs

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

```
1 $ mkdir -p lbthw_heredocs && cd lbthw_heredocs
```

The first line starts with cat followed by a redirection to a file.

Then you use two left chevrons and follow that with a string that represents a marker for the end of the file's contents. In this case, the word is `END`.

```
2 $ cat > afile.txt << END
```

Then you type in whatever you want the file to contain.

```
3 A file can contain
4         whatever you like
```

(The whitespace in the second line above are spaces.)

When you're done, you can finish the file by typing the string you used as a marker on the first line alone on its own line:

```
5 END
```

You can check the contents of the file by just `cat`ing it:

```
6 $ cat afile.txt
```

The marker 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:

```
7  $ function write_echoer_file {
8          cat > $1 << END
9  #!/bin/bash
10 echo $1
11 END
12         chmod +x $1
13 }
14 $ write_echoer_file echoer
15 $ ./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:

```
16 $ function write_echoer_file {
17         cat > $1 << 'END'
18 #!/bin/bash
19 echo $1
20 END
21         chmod +x $1
22 }
23 $ write_echoer_file echoer
24 $ ./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):

```
25 $ function write_echoer_file {
26         cat > $1 <<- 'END'
27                 #!/bin/bash
28                 echo $1
29         END
30         chmod +x $1
31 }
32 $ write_echoer_file echoer
33 $ ./echoer
```

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

```
34 $ function write_echoer_file {
35       cat > $1 <<- 'END'
36             #!/bin/bash
37             echo $1
38       echo Is this the END?
39       END
40       chmod +x $1
41 }
42 $ write_echoer_file echoer
43 $ ./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:

```
44 $ function write_here_string_to_file {
45       cat > $1 <<< $2
46 }
47 $ write_here_string_to_file afile.txt "Write this out"
```

Cleanup

```
48 $ cd .. && rm -rf lbthw_heredocs
```

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.

You will cover:

- Where your history is stored
- How to refer to your previous commands in various ways
- How to record the time your commands were run with the history
- Some options that can be set with the history

How Important is this Section?

Bash's history features are used at the command line so often that it's difficult to understate how important they are. It's a rich subject, but here I keep to the features I use most of the time.

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):

```
1  $ 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:

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

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

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

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

The one I use most often, though, is the second 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 History Shortcuts

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':

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

Starting with the 'bang' sign to indicate we're referring to the history, there follows a colon. Then, you specify the word with a number. The numbers are zero-based, so the arguments start with '1'.

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

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

Here you add a dash 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 `!:$`

You can even reference an argument on the line while you are typing it. Try this:

```
20  $ echo a b !#:1
```

and figure out what's going on. The '`#`' indicates that you are referring to the current line, not the previous one, and the ':1' refers to the second argument (remember, the arguments are numbered from zero, so 'echo' is argument zero.

Finally, another trick I use all the time:

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

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

History Env Vars

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

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

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

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:

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

Was the output of history what you expected? `HISTCONTROL` can determine what gets stored in your history. The directives are separated by colons. Here we use `ignoredups` to tell history to ignore commands that are repeats of the last-recorded command. In the above input, the two consecutive `ls` 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`.

Cleanup

To clean up what you just did:

```
45 $ cd .. && rm -rf lbthw_history
```

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.

How Important is this Section?

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:

```
1  $ function dateit() {
2     while read line
3     do
4       echo "$line $(date '+ %m-%d-%Y %H:%M:%S')"
5     done
6  }
7  $ 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 `vm_stat`, which should be available.

You should be able to follow what's happening there based on what you've learned so far, with the exception of the `while read line` line. `read` is a shell builtin that takes input from standard input until a newline is seen (or an 'end of file' byte on its own line).

```
8  $ read
9  some input
```

It returns a true exit code if input was seen:

```
9  $ echo $?
```

and false if an end of file character is seen (with control-d):

```
10 $ read
11 ^D
12 $ echo $?
```

If an argument is given to read, then a variable is populated with the input:

```
13 $ read myinput
14 some input
15 $ echo $myinput
```

So in the `dateit` function we input above, the while loop keeps taking input from standard input and acts on each line as it's read in, echoing each line to standard output until is gets an end-of-file character, at which point the while loop terminates as read returned a false exit code.

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.

```
16 $ pwd
17 $ 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.

```
18  $ cat > /tmp/lbthwscript.sh << 'EOF'
19  echo My pwd is: $PWD
20  echo I am running in: $(dirname ${BASH_SOURCE[0]})
21  EOF
22  $ chmod +x /tmp/lbthwscript.sh
23  $ /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.

```
24  $ function extract() {
25    if [ -z "$1" ]
26    then
27      echo "Usage: extract <file_name>.<zip|rar|bz2|gz|tar|tbz2|tgz|Z|7z|xz|ex|tar.bz2\
28  |tar.gz|tar.xz>"
29    else
30      if [ -f $1 ] ; then
31         case $1 in
32           *.7z)        7z x $1        ;;
33           *.bz2)       bunzip2 $1     ;;
34           *.exe)       cabextract $1  ;;
35           *.gz)        gunzip $1      ;;
36           *.tar.bz2)   tar xvjf $1    ;;
37           *.tar.gz)    tar xvzf $1    ;;
38           *.tar.xz)    tar xvJf $1    ;;
39           *.tar)       tar xvf $1     ;;
40           *.tbz2)      tar xvjf $1    ;;
```

```
41          *.tgz)       tar xvzf $1    ;;
42          *.Z)         uncompress $1  ;;
43          *.xz)        unxz $1        ;;
44          *.lzma)      unlzma $1      ;;
45          *.rar)       unrar x -ad $1 ;;
46          *.zip)       unzip $1       ;;
47          *)           echo "extract: '$1' - unknown archive method" ;;
48        esac
49     else
50         echo "$1 - file does not exist"
51     fi
52   fi
53 }
54 $ mkdir lbthw_misc
55 $ cd lbthw_misc
56 $ touch a b c
57 $ tar cvfz test.tgz a b c
58 $ rm a b c
59 $ 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:

```
60 $ function sharefiles() {
61       for file in $(ls "$@"); do
62               echo -n $(pwd)
63               [[ $(pwd) != "/" ]] && echo -n /
64               echo $file
65       done
66 }
67 $ 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

```
61 $ cd .. && rm -rf lbthw_misc && 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:

- Signals
- Signal trapping
- Job control
- Debugging techniques, and making your scripts more robust
- String manipulation
- Bash autocomplete

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.

Job Control

In bash, you don't have to wait for the command to complete to start the next one. You can use job control to run processes in the background and manage them.

In this section you'll learn about:

- What job control is
- How to run multiple processes at once in your shell
- How to manage those jobs

How Important is this Section?

Job control is a core feature of bash, and considered a central concept to understand if you are using bash every day.

Starting Jobs

You're going to look at running a simple job using the sleep command.

Type this in:

```
1 $ sleep 60 &
```

You typed a 'normal' command (`sleep 60`) and then added in another character, the ampersand (`&`). The ampersand will run the command 'in the background', which becomes a 'job' in this bash session.

The job has two identifiers, which are immediately reported to you in the terminal.

```
[1] 39165
```

The first is the 'job number', which in this case is '`1`', and the second is the process id, in this case `39165`, but which will be different for you.

If, before the time is up, you run any other commands:

```
2 $ pwd
```

then they are not interfered with by this running job. You can just carry on as normal.

If you wait for the program to finish (60 seconds in this case), and then run any other command, then bash will report to you what happened to that job:

```
3   $ pwd
4   [1]+  Done                    sleep 60
```

Again, it reports the job number, this time with the status (‘Done’), and the command that was originally run (‘sleep 60’).

Controlling Jobs

As well as starting jobs, you can control jobs by sending signals to them.

Here you’re going to start two jobs, one to sleep for two minutes, and the next for one second more (so we can distinguish between them).

```
5   $ sleep 120 &
6   $ sleep 121 &
```

Now you have two jobs running in the background. You can find out what their status is using the `jobs` builtin:

```
1   $ jobs
2   [1]-  Running                 sleep 120 &
3   [2]+  Running                 sleep 121 &
```

Each job is identified by its number in square brackets, followed by a plus or a minus sign. The plus sign indicates which job will be brought to the foreground if you run the `fg` builtin.

Let’s do that here:

```
10  $ fg
11  sleep 121
```

The `sleep 121` process is now running in the foreground. Bash lets you know this by outputting the foregrounded command to the terminal.

Now send the `STOP` signal to the foregrounded process by hitting `\C-z`.

```
[2]+  Stopped                 sleep 121
```

Again, you see the job number, the plus sign, and the status (‘Stopped’). If we want the process to continue to run, then you need to use the `bg` builtin:

```
12 $ bg
13 [2]+ sleep 121 &
```

The process has been continued from its stopped state. If you've taken over two minutes to do all this then you may get a report that the process has terminated with a message that they are '`Done`'. If so, start the two sleep commands again and get back to here.

Now that you have the two sleeps running in the background, you can send other signals to them. For example, you can kill them:

```
14 $ kill %1
15 [1]+  Terminated: 15          sleep 120
```

The per cent sign followed by a number is called a 'job specification' and is the way you can tell bash you want to operate on that job number. Of course, you can also send a signal by using the process identifier as well.

Waiting

One technique that can be useful in bash scripts is to start a number of background processes, and then wait for them to finish before continuing. This can be done with the `wait` builtin.

If your original sleep is still running, then you can run wait.

```
16 $ wait
17 [1]+  Done                    sleep 120
```

Eventually, the wait command returns with a '`Done`' status reported for that process.

Normally, wait always returns an exit code of zero. But if you add a job specification, wait returns the exit code of the last-completed job.

```
18 $ sleep 20 &
19 $ sleep 30 && false &
20 $ wait %1 %2
```

So what exit code will that wait report? Check with:

```
21 $ echo $?
```

What if you swapped the sleep time numbers 20 and 30 above?

What You Learned

- Job control using signals
- Job specifications
- The `fg` and `bg` builtins
- The `wait` builtin

What Next?

In the next section you will build on this knowledge to manage bash processes in the background using job control.

Exercises

1) Write a script to run a series of useful housekeeping or reporting commands in the background, and then output a summary message at the end when they have all completed.

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.

In this section you will learn about:

- What a signal is
- How the `kill` command works
- The `wait` bash builtin
- How to trap signals
- What a 'process group' is

How Important is this Section?

Traps are an advanced concept. If you're new to bash you might want to read this section to be aware of it, and apply it as you get more knowledge of Linux or deeper into bash scripting.

Triggering Signals

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

Follow the instructions here:

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

You should have got the output of a number over 128. 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.

```
3  $ man signal
4  $ 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`. You might not have section 7 of the signal man page installed.

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`.

```
5  $ sleep 999 # NOW HIT CTRL, HOLD IT DOWN AND THEN HIT Z (CTRL-z)
6  $ 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).

```
7  $ sleep 999 &
8  $ KILLPID=$(echo ${!})
9  $ echo ${KILLPID}
10 $ kill -2 ${KILLPID}
11 $ echo ${?}
12 $ wait ${KILLPID}
13 $ 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:

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

Now type out this one and follow the instruction:

```
15  $ mkdir -p lbthw_traps && cd lbthw_traps
16  $ cat > trap_exit.sh << END
17  #!/bin/bash
18  trap "echo trapped" INT
19  while :; do sleep 5; done
20  END
21  $ chmod +x trap_exit.sh
22  $ ./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?

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

Trap Exit

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

Type this out:

```
24  $ cat > trap_exit.sh << END
25  #!/bin/bash
26  trap "echo trapped" EXIT
27  sleep 999 &
28  wait
29  END
30  $ chmod +x trap_exit.sh
31  $ ./trap_exit.sh &
32  $ TRAP_EXIT_PID=$(echo ${!})
33  $ 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:

```
34  $ ./trap_exit.sh &
35  $ TRAP_EXIT_PID=$(echo ${!})
36  $ 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:

```
37 $ ./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 with the `2` (or `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 .. && 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

What Next?

Next we look at various methods used to debug bash scripts.

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 :)

5) Research the other 'special' signal traps. Use `man bash` for this.

Advanced Variables

You covered simple variables off early in this course, but there's more to bash variables than you learned there. In this section you will cover:

- Parameter Expansion
- Associative Arrays

How Important is this Section?

Variables are fundamental to understanding bash commands, much as they are in any programming language. This advanced section takes you through more rarely-used or known aspects of variables. The parameter expansion section is particularly useful to know about.

Parameter Expansion

You've alread seen that you can specify where the variable name starts and ends using curly braces (`${}`).

Bash allows you to do other things with the variable within these curly braces that affect the variable's interpretation by the shell.

These are called 'parameter expansions'. Try this sequence, and figure out what is going on.

```
1 $ unset A
2 $ echo ${A:=123}
3 $ echo $A
```

You can set the variable inline if it's not already set by adding a `:=` after the variable name.

```
4 $ unset A
5 $ echo ${A:-123}
6 $ echo $A
```

When using the `:-` expansion, the 'A' variable is *not* set this time. All that happened is that if `A` is unset, then `123` is substituted in its place.

This is very handy if you want to set defaults, or ensure that a variable is set before continuing with your script.

See what happens when `A` is set by experimenting on the command line. Make sure you understand

Associative Arrays

Bash also supports 'associative arrays'.

> **Note - next commands require bash version 4.x!**
>
> Check this with `echo ${BASH_VERSION}`. If your bash version is below this, then it won't work.
>
> The most likely reason for this is that you are using a Mac. If this is the case, then see: https://apple.stackexchange.com/questions/193411/update-bash-to-version-4-0-on-osx and/or google for how to upgrade to bash version 4. If all else fails, contact the author.

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

```
7   $ declare -A MYAA=([one]=1 [two]=2 [three]=3)
8   $ MYAA[one]="1"
9   $ MYAA[two]="2"
10  $ echo $MYAA
11  $ echo ${MYAA[one]}
12  $ MYAA[one]="1"
13  $ WANT=two
14  $ echo ${MYAA[$WANT]}
```

As well as not being compatible with versions less than 4.0, associative arrays are quite fiddly to create and use, so I don't see them very often in the wild.

What You Learned

In this section you learned about slightly more advanced aspects of bash variables.

You looked at associative arrays, which are common in other scripting languages, but a relatively recent addition to bash (version 4.0+), and also looked at parameter expansion, which is a useful

What Next?

Next you will look at a subject closely related to parameter expansion: string manipulation in bash. You may be surprised at what bash is capable of.

Exercises

1) Search in the `man` page for 'Parameter Expansion' and experiment with all the different types available.

2) Create an associative array to map color names to terminal escape codes for text colors. Write a script to use the array to output text in a given color based on a command line argument.

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!

In this section you'll cover:

- How to edit strings in bash
- What extglobs are and how you can use them
- How to avoid common quoting problems

How Important is this Section?

This section is not essential. String manipulation in bash is occasionally useful to know about, but Linux offers many more well-known to do these jobs as well.

The quoting tricks in this section are occasionally very useful, however.

String Length

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

```
1 $ A='12345678901234567890'
2 $ echo "${#A}"
3 $ 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 explains how to parse *n* characters starting from a particular position.

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

```
4 $ echo ${A:2}
5 $ 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:

```
6 $ echo "${A/234/432}"
7 $ echo "${A//234/432}"
```

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

String Editing

Another commonly-required string operation is getting a substring.

This outputs everything from the third character in the string:

```
8 $ echo ${A:2}
```

and this outputs two numbers from the fourth character in the string:

```
9 $ echo ${A:3:2}
```

so you can tell from this that the strings are `0`-indexed, that is the first character is numbered zero, the second is numbered one, and so on.

You can also search and replace within strings:

```
10 $ B=0000000000
11 $ echo ${B/0/1}
```

The simple pattern replaces only the first zero with a `1`. If you want the pattern to go across all the matches in the string, then add another slash:

```
12 $ echo ${B//0/1}
```

Note - next commands require bash version 4.x!

Check this with `echo ${BASH_VERSION}`. If your bash version is below this, then it won't work.

The most likely reason for this is that you are using a Mac. If this is the case, then see: https://apple.stackexchange.com/questions/193411/update-bash-to-version-4-0-on-osx and/or google for how to upgrade to bash version 4. If all else fails, contact the author.

You can also convert lower case to upper case and vice versa. Using a single comma will turn the starting title case character into a lower case character:

```
13  $ C=AbCd
14  $ echo ${C,}
```

whereas using two will lower-case the entire string:

```
15  $ echo ${C,,}
16  $ C=aAbCd
```

Similarly, the caret character will upper-case a string also:

```
17  $ echo ${C^}
18  $ echo ${C^^}
```

And More…

There are many more string manipulation tricks you can use, but it is not worth trying to learn them all in one go.

If you want to study them in more detail and find out what's available (or ever find yourself wondering what can be done in bash with strings) then have a look at the man page under 'Parameter Expansion'.

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 those too!). Feel free to skip this section if you feel it's too obscure.

```
19  $ shopt -s extglob
20  $ A="12345678901234567890"
21  $ 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.

```
25 $ echo "B      |${B}|"
26 $ echo "B#+( ) |${B#+( )}|"
27 $ echo "B#?( ) |${B#?( )}|"
28 $ echo "B#*( ) |${B#\*( )}|"
29 $ echo "B##+( )|${B##+( )}|"
30 $ echo "B##*( )|${B##*( )}|"
31 $ 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:

```
32 $ TODAY=$(date +%j)
33 $ 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:

```
34 $ 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:

```
35 $ 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:

```
36 $ 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

No cleanup required in this section, though you may want to quit your bash shell and start a fresh one, as you set a shell option (extglob) above.

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

What Next

In the next section you will look at bash's debugging capabilities.

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!'

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.

In this section you will learn about:

- Various bash flags useful for debugging
- How to trace bash code to determine what it's doing
- Linting scripts in bash

How Important is this Section?

If you write, use or maintain bash of any complexity you'll want to know how to debug it!

Syntax Checking Options

Start by creating this simple script:

```
1 $ mkdir -p lbthw_debugging && cd lbthw_debugging
2 $ cat > debug_script.sh << 'END'
3 #!/bin/bash
4 A=some value
5 echo "${A}
6 echo "${B}"
7 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.

```
8 $ bash -n debug_script.sh
```

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

```
9 $ bash debug_script.sh
```

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

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

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

```
11  $ 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.

Fix the error you see before continuing.

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.

```
12  #!/bin/bash
13  set -o nounset
14  A="some value"
15  echo "${A}"
16  echo "${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?

```
17  #!/bin/bash
18  set -o nounset
19  A="some value"
20  B=
21  echo "${A}"
22  echo "${B}"
```

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

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:

```
23  #!/bin/bash
24  set -o nounset
25  set -o xtrace
26  declare A="some value"
27  PS4='$(date "+%s%N => ")'
28  B=
29  echo "${A}"
30  A="another value"
31  echo "${A}"
32  echo "${B}"
33  ls
34  pwd
35  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 and wo\
rd splitting.


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

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


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


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 use '\
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 use '\
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:

```
36 $ 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 shellcheck to help you reduce the risk of your scripts failing

What Next

Next you will look at autocomplete functionality in bash.

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.

Autocomplete

If you've used bash for any length of time, then you will likely already be familiar with its autocomplete capabilities.

If not, then you might want to start using it!

In this section you will cover:

- How autocomplete works in bash
- The bash `shift` command
- Changing case using string manipulation

> ## Note - requires bash version 4.x!
>
> Check this with `echo ${BASH_VERSION}`. If your bash version is below this, then it won't work.
>
> The most likely reason for this is that you are using a Mac. If this is the case, then see: https://apple.stackexchange.com/questions/193411/update-bash-to-version-4-0-on-osx and/or google for how to upgrade to bash version 4. If all else fails, contact the author.

How Important is this Section?

In this section you will build on what you learned previously about:

- Bash arrays
- Terminal escape codes
- Built-ins
- Bash functions
- Bash startup scripts

to comprehend more deeply a feature you will probably use almost every time you use bash.

Autocompleting Commands

If you hit the 'Tab' key on your keyboard twice at the default prompt, then you should see something like this:

```
Display all 2266 possibilities (y or n)?
```

If you're brave enough to hit 'y' then you will see a list of all the commands available in your path.

To escape that list, hit 'q'.

Now if you try typing 'z' on a fresh shell command line, and then hit 'Tab' twice, you will likely immediately see the commands available that begin with 'z'.

That's autocomplete in bash - by default, you get the commands available that match the letters you've typed in so far. This is handy if you can only remember part of a command, or you just want to see what commands start with 'q' (try it!).

Autocompleting Arguments

In addition to looking for commands, bash's autocomplete can be made to be context-aware. A simple example of this is to see what happens when you type `ls`, and then a space, and then 'Tab' twice on a fresh shell line. What happens? You should see that this time commands are not listed, but something else…

Listing local files is something of a default in bash - it assumes that arguments are likely to be files. What I'm going to show you next is how you can get bash to be even more context dependent with auto-complete to the point where you can advise the user of your specific program about what is available to them.

A Simple Program

First you're going to create a program called `myecho`. As you type it in, try and work out what it's doing.

If you don't understand a line or command in there, then try and find out what it does by playing with it in a toy script, or try and find out from `man bash`.

```
1  $ mkdir -p lbthw_autocomplete && cd lbthw_autocomplete
2  $ cat > myecho << END
3  > #!/usr/bin/env bash
4  > function usage() {
5  >         echo "Usage: $0 [red|green|blue] [message]"
6  >         exit $1
7  > }
8  > if [ -z "$1" ]
9  > then
10 >         usage 1
11 > fi
12 > typeset -l COLOR="$1"
```

```
13  > shift
14  > RED='\033[0;31m'
15  > GREEN='\033[0;32m'
16  > BLUE='\033[0;34m'
17  > MSG="$@"
18  > COLOR=${COLOR,,}
19  > if [[ "${COLOR}" = 'red' ]]
20  > then
21  >         MSG="${RED}${@}${RED}"
22  > elif [[ "${COLOR}" = 'green' ]]
23  > then
24  >         MSG="${GREEN}${@}${GREEN}"
25  > elif [[ "${COLOR}" = 'blue' ]]
26  > then
27  >         MSG="${BLUE}${@}${BLUE}"
28  > else
29  >         usage 1
30  > fi
31  > echo -e "${MSG}"
32  > END
```

If you're still struggling, then other sections of this book will help enlighten you. The only new command here is `shift`. You may be able to work out what `shift` does from context, but just in case it's not obvious, it takes the first argument from your command line and removes it from the list of arguments passed in. This is handy for processing command line arguments, as you can keep calling `shift` until all the arguments are processed. Such a looping construct is used in many bash scripts.

You may also have not come across the variable `$@` before. Try finding out from the `man bash` page what it holds.

If you can't find it, don't worry, it's not easy to find in there. Look for the section on 'Special Parameters'.

If you still can't figure it out, then try adding echo statements to figure out what's going on. Keep at it.

Finally, can you figure out what the two commas in `${COLOR,,}` do? If not, play with it on the command line. This playing is how you learn and embed the bash knowledge as you get more advanced.

Now you've created the `myecho` script, make it executable and available on your path, and run it:

```
33 $ chmod +x myecho
34 $ PATH=.:${PATH}
35 $ myecho red WARNING this is dangerous
36 $ myecho green OK good to go
```

Try changing the case of the color argument (eg 'GreeN' and see what happens. Was that what you expected?). What part of the code is responsible for this?

Adding Autocomplete Functionality

Now let's say that it is some time later, and you've forgotten about your brilliant `myecho` script.

You type `myecho` on the command line, followed by a space and then hit 'Tab' twice.

Wouldn't it be good if autocomplete could present the three color options to you when autocompleting?

You can do that. Type out this script:

```
37 $ cat > myechocomplete << END
38 > myecho_completion()
39 > {
40 >   COMPREPLY+=("red")
41 >   COMPREPLY+=("blue")
42 >   COMPREPLY+=("green")
43 > }
44 > complete -F myecho_completion myecho
45 > END
```

To make your shell aware of it, type:

```
46 $ source myechocomplete
```

Now if you type `myecho`, followed by a space and then two 'Tab's, you should see the three options.

The `complete` command is a builtin, and the `-F` flag expects a function to be given to it, and will run it when the last argument to complete is typed on the command line and autocomplete is triggered.

When the function is run bash expects you to add to an array called `COMPREPLY` to give the options that should be made available to the user when they try to autocomplete. The three colors added above are the displayed to the user from the array.

Advanced Autocomplete Functionality

If you want to get a bit more sophisticated with bash's autocomplete, then you have other variables than `COMPREPLY` available to you. For this simple program you're going to provide a hint to input a message if the user has already input a color argument to `myecho`.

```
47  $ cat > myechocomplete << END
48  > myecho_completion()
49  > {
50  >   if [ ${COMP_CWORDS -ge 2 ]
51  >   then
52  >     COMPREPLY+=("Input")
53  >     COMPREPLY+=("message!")
54  >   else
55  >     COMPREPLY+=("red")
56  >     COMPREPLY+=("blue")
57  >     COMPREPLY+=("green")
58  >   fi
59  > }
60  > complete -F myecho_completion myecho
61  > END
62  $ source myechocomplete
```

Now if you type `myecho red`, followed by a space and then two 'Tab's, you should see the instruction on the line.

This is not exactly a neat way to get this information across. Unfortunately, the completion output functionality is not very flexible. You might want to try implementing a better solution with `echo`s or longer `COMPREPLY` entries to see its limitations.

Automate It

Although this is useful, it is not realistic to `source` the completion scripts every time you might want to use the particular command you have an autocomplete script for.

For this reason, you can add these functions to your bash startup scripts (see the section on 'Scripts and Startups' in Part I) so that they are available whenever you use bash on that machine.

Cleanup

To clean up, run:

```
63  $ cd .. && rm -rf lbthw_autocomplete
```

What You Learned

- How autocomplete works in bash
- How to create your own autocomplete function
- How to handle arguments to bash scripts and functions using `shift`
- How to change the case of string variables

What Next?

To finish this book, you will look at a more complete bash program that can function as a continuous integration tool.

Exercises

1) Write autocomplete scripts for already-existing programs you use often.

2) Look for autocomplete scripts on GitHub, and install and use them locally.

3) Read the bash manual section and figure out what all the `COMP_` variables store when autocompleting. Write example programs to use them.

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 practical context.

How Important is this Section?

This section is not vital, but it may be interesting to see what a substantial script looks like.

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:

```
1   #!/bin/bash
2
3   # Options are placed right at the top.
4   set -o pipefail
5   set -o errunset
6
7   # Set up defaults to sensible values.
8   FORCE=0
9   VERBOSE=0
10  REPO=""
11  EMAIL=""
12  NAME="ci"
13  TEST_DIR="."
14  TEST_COMMAND="./test.sh"
15  MAIL_CMD="mail"
16  MAIL_CMD_ATTACH_FLAG="-A"
17  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 needed)\
40  (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=${NAME\
48 })
49 -d - directory within repository to navigate to (default=${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}, empty s\
52 tring 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 fla\
55 g 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 that)
59 -h - show help
60 
```

```
61  EXAMPLES
62
63  - "Clone -r https://github.com/ianmiell/shutit.git if a git pull on /space/git/shuti\
64  t indicates there's been an update.
65    Then navigate to test, run ./test.sh and mail ian@mail.meirionconsulting.com if th\
66  ere are any issues"
67
68    ./cheapci -r https://github.com/ianmiell/shutit.git -l /space/git/shutit -d test -\
69  c ./test.sh -m ian@mail.meirionconsulting.com
70
71
72  - "Run this continuously in a crontab."
73
74    Crontab line:
75
76    * * * * * cd /path/to/cheapci && ./cheapci -r https://github.com/ianmiell/shutit.g\
77  it -l /space/git/shutit -d test -c ./test.sh -m ian@mail.meirionconsulting.com
78  END
79  }
80
81  # Next we gather options that were passed in from the command line. Do you know
82  # what the ':'s and '?' symbols mean below?
83  while getopts "h?vfm:n:d:r:l:c:a:q:w:t:e:s:" opt
84  do
85          case "$opt" in
86          h|\?)
87                  show_help
88                  exit 0
89                  ;;
90          v) VERBOSE=1 ;;
91          f) FORCE=1 ;;
92          r) REPO=$OPTARG ;;
93          m) EMAIL=$OPTARG ;;
94          n) NAME=$OPTARG ;;
95          d) TEST_DIR=$OPTARG ;;
96          l) LOCAL_CHECKOUT=$OPTARG ;;
97          c) TEST_COMMAND=$OPTARG ;;
98          a) MAIL_CMD=$OPTARG ;;
99          q) PRE_SCRIPT=$OPTARG ;;
100         w) POST_SCRIPT=$OPTARG ;;
101         a) MAIL_CMD=$OPTARG ;;
102         t) MAIL_CMD_ATTACH_FLAG=$OPTARG ;;
103         e) MAIL_CMD_RECIPIENTS_FLAG=$OPTARG ;;
```

```
104         s) MAIL_CMD_SUBJECT_FLAG=$OPTARG ;;
105         i) TIMEOUT_S=$OPTARG ;;
106         esac
107 done
108 
109 # This line 'shifts' all the arguments off the command. The 'OPTIND' variable
110 # contains the number of options found by getopts.
111 shift "$((OPTIND-1))"
112 
113 # We require that the git REPO variable is set up by this point, so if it is
114 # not, then we show the help and exit with a non-zero code to show that the
115 # run did not complete successfully.
116 if [[ ${REPO} = "" ]]
117 then
118         show_help
119         exit 1
120 fi
121 
122 # Rather than set xtrace on every time, we set it on only if the verbose flag
123 # was used.
124 if [[ ${VERBOSE} -gt 0 ]]
125 then
126         set -x
127 fi
128 
129 # More variables, this time derived from the optional values.
130 # Create variables for items that will be re-used rather than using 'magic
131 # values'.
132 BUILD_DIR_BASE="/tmp/${NAME}"
133 BUILD_DIR="${BUILD_DIR_BASE}/${NAME}_builddir"
134 mkdir -p "${BUILD_DIR}"
135 # Use of the RANDOM variable to create a log file hopefully unique to this run.
136 LOG_FILE="${BUILD_DIR}/${NAME}_build_${RANDOM}.log.txt"
137 BUILD_LOG_FILE="${BUILD_DIR}/${NAME}_build.log.txt"
138 # Create a lock file based on the name given
139 LOCK_FILE="${BUILD_DIR}/${NAME}_ci.lck"
140 
141 # Create a generic cleanup function in case it is needed laster
142 function cleanup() {
143         rm -rf "${BUILD_DIR}"
144         rm -f "${LOCK_FILE}"
145         # get rid of /tmp detritus, leaving anything accessed 2 days ago+
146         find "${BUILD_DIR_BASE}"/* -type d -atime +1 | rm -rf
```

```
147         echo "cleanup done"
148 }
149 
150 # Trap specific signals and run cleanup
151 trap cleanup TERM INT QUIT
152 
153 # Function to send mail. Note the use of the array log_file_arg in the mail
154 # command.
155 function send_mail() {
156         msg=${1}
157         if [[ ${LOG_FILE} != "" ]] && [[ ${MAIL_CMD_ATTACH_FLAG} != "" ]]
158         then
159                 log_file_arg=(${MAIL_CMD_ATTACH_FLAG} ${LOG_FILE})
160         fi
161         if [[ ${EMAIL} != "" ]] && [[ ${MAIL_CMD} != "" ]]
162         then
163                 echo "${msg}" | ${MAIL_CMD} "${MAIL_CMD_SUBJECT_FLAG}" "${msg}" "${l\
164 og_file_arg[@]}" "${MAIL_CMD_RECIPIENTS_FLAG}" "${EMAIL}"
165         else
166                 echo "${msg}"
167         fi
168 }
169 
170 # Output the date to the log file.
171 date 2>&1 | tee -a "${BUILD_LOG_FILE}"
172 
173 # Use the -a test to determine whether this ci is currently running.
174 if [[ -a ${LOCK_FILE} ]]
175 then
176         echo "Already running" | tee -a "${BUILD_LOG_FILE}"
177         exit
178 else
179         touch "${LOCK_FILE}"
180         # Fetch changes from the remote servers.
181         pushd "${LOCAL_CHECKOUT}"
182         git fetch origin master 2>&1 | tee -a "${BUILD_LOG_FILE}"
183         # See if there are any incoming changes
184         updates=$(git log HEAD..origin/master --oneline | wc -l)
185         echo "Updates: ${updates}" | tee -a "${BUILD_LOG_FILE}"
186         if [[ ${updates} -gt 0 ]] || [[ ${FORCE} -gt 0 ]]
187         then
188                 touch "${LOG_FILE}"
189                 pushd "${LOCAL_CHECKOUT}"
```

```
190                 echo "Pulling" | tee -a "${LOG_FILE}"
191                 git pull origin master 2>&1 | tee -a "${LOG_FILE}"
192                 popd
193                 # This won't exist in a bit so no point pushd'ing
194                 pushd "${BUILD_DIR}"
195                 # Clone to NAME
196                 git clone "${REPO}" "${NAME}"
197                 popd
198                 ${PRE_SCRIPT} 2>&1 | tee -a "${LOG_FILE}"
199                 EXIT_CODE="${?}"
200                 if [[ ${EXIT_CODE} -ne 0 ]]
201                 then
202                         msg="ANGRY ${NAME} on $(hostname)"
203                 fi
204                 pushd "${BUILD_DIR}"/"${NAME}"/"${TEST_DIR}"
205                 timeout "${TIMEOUT_S}" "${TEST_COMMAND}" 2>&1 | tee -a "${LOG_FILE}"
206                 EXIT_CODE=$?
207                 popd
208                 if [[ ${EXIT_CODE} -ne 0 ]]
209                 then
210                         if [[ ${EXIT_CODE} -eq 124 ]]
211                         then
212                                 msg="ANGRY (TIMEOUT) ${NAME} on $(hostname)"
213                         else
214                                 msg="ANGRY ${NAME} on $(hostname)"
215                         fi
216                 else
217                         msg="HAPPY ${NAME} on $(hostname)"
218                 fi
219                 ${POST_SCRIPT} 2>&1 | tee -a "${LOG_FILE}"
220                 EXIT_CODE=$?
221             if [[ ${EXIT_CODE} -ne 0 ]]
222                 then
223                         msg="ANGRY ${NAME} on $(hostname)"
224                 fi
225                 send_mail "${msg}"
226         fi
227         cleanup
228 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
- Able to 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 any bash issue you come across.

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

I welcome feedback and comment to ian@mail.meirionconsulting.com or on Twitter: @ianmiell

Note also that I offer training in Git, Bash, and Docker to anyone interested. Just get in touch if you want to know more.