Mastering Linux Shell Scripting

Shell scripting is a quick method to prototype a complex application or a problem. A shell script is a command or a collection of commands to automate tasks. When working on Linux-based systems using simple one-line commands or a sequence of them in the shell, complex problems can be solved with greater ease and we can perform tasks such as text processing or backing up sysadmin tools.

Starting from the basics, we explain each step of creating great code. We explain creating and running script in normal mode and in debug mode. You will then learn about various conditional statements' code snippets. Furthermore, we also cover how to implement functions and edit files using the Stream Editor, script in Perl, and program in Python. The book will capture your attention and keep you engaged with the simplicity and clarity of each explanation. Every step is accompanied with screen captures so you can cross-check the results before moving on.

Who this book is written for

Mastering Linux Shell Scripting has been written for Linux administrators who want to automate tasks in their daily lives, saving time and effort. You need to have command-line experience and be familiar with the tasks that you need to automate. Basic knowledge of scripting is expected.

What you will learn from this book

- Use the type command to identify the order of command evaluation
- Realize that elements, such as $, which you have used in test conditions, are commands
- Create interactive scripts that prompt for user input
- Foster menu structures for operators with little command-line experience
- Develop scripts that dynamically edit web configuration files to produce a new virtual host
- Write scripts that use Awk to search and report on log files
- Draft effective scripts using functions as building blocks, reducing maintenance and build time
- Make informed choices by comparing different script languages such as Perl and Python with Bash
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'What and Why of Scripting with Bash'
- A synopsis of the book’s content
- More information on Mastering Linux Shell Scripting
About the Author

Andrew Mallett is the owner of The Urban Penguin and is a comprehensive provider for professional Linux software development, training, and services. Having always been a command-line fan, he feels that so much time can be saved through knowing command-line shortcuts and scripting. TheUrbanPenguin YouTube channel, maintained by Andrew, has well over 800 videos to support this and he has authored four other Packt titles.
Preface

*Mastering Linux Shell Scripting* will become your Bible and a handbook to create and edit bash shell scripts in Linux, OS X, or Unix. Starting with the fundamentals, we quickly move onto helping you create useful scripts with practical examples. In this way, your learning becomes effective and quick. With each chapter, we provide explanations of the code and code examples, so from a learning book this becomes a book that you can use as a ready reference, if you need to understand how to program a specific task.

**What this book covers**

*Chapter 1, What and Why of Scripting with Bash*, explains how to create and name scripts. Once you have the script created you can make it executable and welcome yourself to the world. If you have little or no knowledge of scripting then you can start here.

*Chapter 2, Creating Interactive Scripts*, covers the scripts we will need to work in a more flexible manner and accept arguments or even prompt the user for input during the script execution. I am sure that you have seen scripts similar to this asking for installation directories or server's tp connect to.

*Chapter 3, Conditions Attached*, covers the use of keywords, such as "if", and commands like "test". It tells us how we can start creating decision structures in the code and then prompt the user for input, if we have not provided arguments; otherwise, we can run silently.

*Chapter 4, Creating Code Snippets*, covers the vim text editor, which is very powerful and also syntax highlighting to help us edit the script. However, we can also read into files of the current script. In this way, we can create snippets of code to represent commonly used blocks.
Chapter 5, *Alternative Syntax*, tells us how we can abbreviate the test command to just a single `, we can also use `[[` and `((` depending on your needs.

Chapter 6, *Iterating with Loops*, covers how loops are also conditional statements. We can repeat a block of code while a condition is true or false. In this way, using for, while, or until we can have the script complete the repetitive code sequences.

Chapter 7, *Creating Building Blocks with Functions*, covers how functions can encapsulate the code that we need to repeat within the script. This can help with readability and how easy a script is to maintain.

Chapter 8, *Introducing sed*, the stream editor, tells us how sed can be used to edit files dynamically and implement it in scripts. In this chapter, we look at how to use and work with sed.

Chapter 9, *Automating Apache Virtual Hosts*, covers the practical recipes that we can take away when we create a script to create virtual hosts on an Apache HTTPD server. We use sed within the scripts to edit the template used to define virtual hosts.

Chapter 10, *Awk Fundamentals*, looks at how we can start to process text data from the command line and using awk is another very powerful tool in Linux.

Chapter 11, *Summarizing Logs with Awk*, tells us about the first practical example we look at with awk, allowing us to process log files on the web server. It also looks at how to report the IP address that has access to the server most often, as well as, how many errors occur and of which type.

Chapter 12, *A Better lastlog with Awk*, looks at more examples that we can use in awk to filter and format data provided by the lastlog command. It drills down to the specific information that we want and removes information we do not need.

Chapter 13, *Using Perl as a Bash Scripting Alternative*, introduces the Perl scripting language and the advantages that it can offer. We are not restricted to just using bash we also have Perl as a scripting language.

Chapter 14, *Using Python as a Bash Scripting Alternative*, introduces you to Python and the Zen of Python that will help you with all programming languages. Like Perl, Python is a scripting language that can extend the functionality of your scripts.
Welcome to the what and why of bash scripting. My name is Andrew Mallett and I am a bash scripting junkie or perhaps more accurately: a scripting junkie. As an administrator, I fail to see the need to do repetitive tasks manually. We get time for more interesting things when we choose scripts to carry out the laborious tasks that we don't like. In this chapter, we will introduce you to the what and why of bash scripting. If you are new, it will help you become familiar with scripts and also provide some great insights for those with more experience and who want to improve their skills. As we make our way through the chapter, each element is designed to be added to your knowledge to help you achieve your goals. While doing so, we will be covering the following topics:

- Bash vulnerabilities
- The bash command hierarchy
- Preparing text editors for scripting
- Creating and executing scripts
- Debugging your scripts

**Bash vulnerabilities**

For this book, I will be working entirely on a Raspberry Pi 2 running Raspbian, a Linux distribution similar to Debian, and Ubuntu; although for you, the operating system you choose to work with is immaterial, in reality, as is the version of bash. The bash version I am using is 4.2.37(1). If you are using the OS X operating system, the default command line environment is `bash`. 
To return the operating system being used, type the following command if it is installed:

$ lsb_release -a

The output from my system is shown in the following screenshot:

```
No LSB modules are available.
Distributor ID: Debian
Description: Debian GNU/Linux 7.8 (wheezy)
Release: 7.8
Codename: wheezy
pi@pilabs ~ $ ...
```

The easiest way to determine the version of bash that you are using is to print the value of a variable. The following command will display your bash version:

$ echo $BASH_VERSION

The following screenshot displays the output from my system:

```
pi@pilabs ~ $ echo $BASH_VERSION
4.2.37(1)-release
pi@pilabs ~ $ ...
```

In 2014, there was a well-publicized bug within bash that had been there for many years—the shell-shock bug. If your system is kept up-to-date, then it is not likely to be an issue but it is worth checking. The bug allows malicious code to be executed from within a malformed function. As a standard user, you can run the following code to test for the vulnerabilities on your system. This code comes from Red Hat and is not malicious but if you are unsure then please seek advice.

The following is the code from Red Hat to test for the vulnerability:

```
$ env 'x=() { }; echo vulnerable''BASH_FUNC_x()=() { }; echo vulnerable' bash -c "echo test"
```

If your system is free from this first vulnerability the output should be as shown in the following screenshot:

```
bash: warning: x: ignoring function definition attempt
bash: error importing function definition for 'BASH_FUNC_x'
test
```

---

[2]
To test for the last vulnerability from this bug, we can use the following test, which is again from Red Hat:

```bash
cd /tmp; rm -f /tmp/echo; env 'x=() { (a)=>\'' bash -c "echo date"; cat /tmp/echo
```

The output from a patched version of bash should look like the following screenshot:

```
date
```

```
cat: /tmp/echo: No such file or directory
```

If the output from either of these command lines is different, then your system may be vulnerable to shell-shock and I would update bash or at least take further advice from a security professional.

### The bash command hierarchy

When working on at the bash shell and when you are sitting comfortably at your prompt eagerly waiting to type a command, you will most likely feel that it is a simple matter of typing and hitting the `Enter` key. You should know better than to think that things are never quite as simple as we imagine.

#### Command type

For example, if we type and enter `ls` to list files, it will be reasonable to think that we were running the command. It is possible, but we will be running an alias often. Aliases exist in memory as a shortcut to commands or commands with options; these aliases are used before we even check for the file. The bash shell built-in command `type` can come to our aid here. The `type` command will display the type of command for a given word entered at the command line. The types of command is listed as follows:

- Alias
- Function
- Shell built in
- Keyword
- File

This list is also a representative of the order in which they are searched. As we can see, it is not until the very end where we search for the executable file `ls`.
The following command demonstrates the simple use type:

```
$ type ls
ls is aliased to `ls --color=auto`
```

We can extend this further to display all the matches for the given command:

```
$ type -a ls
ls is aliased to `ls --color=auto'
ls is /bin/ls
```

If we need to just type in the output, we can use the `-t` option. This is useful when we need to test the command type from within a script and only need the type to be returned. This excludes the superfluous information; thus, making it easier for us humans to read. Consider the following command and output:

```
$ type -t ls
alias
```

The output is clear and simple and just what a computer or script requires.

The built-in `type` can also be used to identify shell keywords such as if, case, function, and so on. The following command shows type being used against multiple arguments and types:

```
$ type ls quote pwd do id
```

The output of the command is shown in the following screenshot:

```
pi@pilabs /tmp $ type ls quote pwd do id
ls is aliased to `ls --color=auto'
quote is a function
quote ()
{
    local quoted=${1//'/"\\\\\\\''};
    printf "\%s" "$quoted"
}
pwd is a shell builtin
do is a shell keyword
id is /usr/bin/id
pi@pilabs /tmp $ _
```

You can also see that the function definition is printed when we stumble across a function when using `type`. 
Command PATH

Linux will check for executables in the PATH environment only when the full or relative path to the program is supplied. In general, the current directory is not searched unless it is in the PATH. It is possible to include our current directory within the PATH by adding the directory to the PATH variable. This is shown in the following code example:

```
$ export PATH=$PATH:.
```

This appends the current directory to the value of the PATH variable each item the PATH is separated using the colon. Now, your PATH is updated to include the current working directory and each time you change directories, the scripts can be executed easily. In general, organizing scripts into a structured directory hierarchy is probably a great idea. Consider creating a subdirectory called bin within your home directory and add the scripts into that folder. Adding $HOME/bin to your PATH variable will enable you to find the scripts by name and without the file path.

The following command-line list will only create the directory, if it does not already exist:

```
$ test -d $HOME/bin || mkdir $HOME/bin
```

Although the above command-line list is not strictly necessary, it does show that scripting in bash is not limited to the actual script and we can use conditional statements and other syntax directly at the command line. From our viewpoint, we know that the preceding command will work whether you have the bin directory or not. The use of the $HOME variable ensures that the command will work without considering your current file system context.

As we work through the book, we will add scripts into the $HOME/bin directory so that they can be executed regardless of our working directory.

Preparing text editors for scripting

Throughout the book, I will be working on the command line of Raspberry Pi and this will include the creation and editing of the scripts. You, of course, can choose the way you wish to edit your script and may prefer to make use of a graphical editor and I will show some settings in gedit. I will make one excursion to a Red Hat system to show screenshots of gedit in this chapter.

To help make the command line editor easier to use, we can enable options and we can persist with these options through hidden configuration files. The gedit and other GUI editors and their menus will provide similar functionality.
Configuring vim

Editing the command line is often a must and is a part of my everyday life. Setting up common options that make life easier in the editor give us the reliability and consistency you need, a little like scripting itself. We will set some useful options in the vi or vim editor file, $HOME/.vimrc.

The options we set are detailed in the following list:

- **showmode**: Ensures we see when we are in insert mode
- **nohlsearch**: Does not highlight the words that we have searched for
- **autoindent**: We indent our code often; this allows us to return to the last indent level rather than the start of a new line on each carriage return
- **tabstop=4**: Sets a tab to be four spaces
- **expandtab**: Converts tabs to spaces, which is useful when the file moves to other systems
- **syntax on**: Note that this does not use the set command and is used to turn on syntax highlighting

When these options are set, the $HOME/.vimrc file should look similar to this:

```
set showmode
set nohlsearch
set autoindent
set tabstop=4
set expandtab
syntax on
```

Configuring nano

The nano text edit is increasing in importance and it is the default editor in many systems. Personally, I don't like the navigation or the lack of navigation features that it has. It can be customized in the same way as vim. This time we will edit the $HOME/.nanorc file. Your edited file should look something like the following:

```
set autoindent
set tabsize 4
include /usr/share/nano/sh.nanorc
```

The last line enables syntax highlighting for shell scripts.
Configuring gedit

Graphical editors, such as gedit, can be configured using the preferences menu and are pretty straightforward.

Enabling tab spacing to be set to 4 spaces and expanding tabs to spaces can be done using the Preference | Editor tab, as shown in the following screenshot:

![gedit Preferences](image)

---

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Another very useful feature is found on the Preferences | Plugins tab. Here, we can enable the Snippets plugin that can be used to insert code samples. This is shown in the following screenshot:

For the rest of the book, we will be working on the command line in and in vim; feel free to use the editor that you work with best. We have now laid the foundations to create good scripts and although whitespace, tabs, and spaces in bash scripts are not significant; a well laid out file with consistent spacing is easy to read. When we look at Python later in the book, you will realize that in some languages the whitespace is significant to the language and it is better to adopt the good habits early.

Creating and executing scripts

With our editors primed and ready, we can now move quickly to creating and executing our scripts. If you are reading this book with some prior experience, I will warn you that we are going to start with the basics but we will also include looking at positional parameters; feel free to move on at your own pace.
Hello World!
As you know, it is almost obligatory to begin with a hello world script and we will not disappoint as far as this is concerned. We will begin by creating a new script $HOME/bin/hello1.sh. The contents of the file should read as in the following screenshot:

```
#!/bin/bash
echo "Hello World"
exit 0
```

I am hoping that you haven't struggled with this too much; it is just three lines after all. I encourage you to run through the examples as you read to really help you instill the information with a good hands-on practice.

- `#!/bin/bash`: Normally, this is always the first line of the script and is known as the shebang. The shebang starts with a comment but the system still uses this line. A comment in a shell script has the `#` symbol. The shebang instructs the system to the interpreter to execute the script. We use bash for shell scripts and we may use PHP or Perl for other scripts, as required. If we do not add this line, then the commands will be run within the current shell; it may cause issues if we run another shell.
- `echo "Hello World"`: The echo command will be picked up in a built-in shell and can be used to write a standard output, STDOUT, this defaults to the screen. The information to print is enclosed in double-quotes, there will be more on quotes later.
- `exit 0`: The exit command is a built in shell and is used to leave or exit the script. The exit code is supplied as an integer argument. A value of anything other than 0 will indicate some type of error in the script's execution.

Executing the script
With the script saved in our PATH environment, it still will not execute as a standalone script. We will have to assign and execute permissions for the file, as needed. For a simple test, we can run the file directly with bash. The following command shows you how to do this:

```
$ bash $HOME/bin/hello1.sh
```
What and Why of Scripting with Bash

We should be rewarded with the Hello World text being displayed back on our screens. This is not a long-term solution, as we need to have the script in the $HOME/bin directory, specifically, to make the running of the script easy from any location without typing the full path. We need to add in the execute permissions as shown in the following code:

```bash
$ chmod +x $HOME/bin/hello1.sh
```

We should now be able to run the script simply, as shown in the following screenshot:

<table>
<thead>
<tr>
<th>pi@pilabs ~</th>
<th>$ chmod +x $HOME/bin/hello1.sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi@pilabs ~</td>
<td>$ hello1.sh</td>
</tr>
<tr>
<td>pi@pilabs ~</td>
<td>Hello World</td>
</tr>
<tr>
<td>pi@pilabs ~</td>
<td>$ _</td>
</tr>
</tbody>
</table>

Checking the exit status

This script is simple but we still have to know how to make use of the exit codes from scripts and other applications. The command-line list that we generated earlier while creating the $HOME/bin directory, is a good example of how we can use the exit code:

```bash
$ command1 || command2
```

In the preceding example, command2 is executed only if command1 fails in some way. To be specific, command2 will run if command1 exits with a status code other than 0.

Similarly, in the following extract:

```bash
$ command1 && command2
```

We will only execute command2 if command1 succeeds and issues an exit code of 0.

To read the exit code from our script explicitly, we can view the $? variable, as shown in the following example:

```bash
$ hello1.sh
$ echo $?
```

The expected output is 0, as this is what we have added to the last line of the file and there is precious little else that can go wrong causing us to fail in reaching that line.
Ensuring a unique name

We can now create and execute a simple script but we need to consider the name a little. In this case, `hello1.sh` is going to be good enough and is unlikely to clash with anything else on the system. We should avoid using names that may clash with existing aliases, functions, keywords, and building commands, as well as, avoid names of programs already in use.

Adding the `sh` suffix to the file does not guarantee the name to be unique but in Linux, where we do not file extensions, the suffix is a part of the file name. This helps you to provide a unique identity to your script. Additionally, the suffix is used by the editor to help you identify the file for syntax highlighting. If you recall, we specifically added the syntax highlighting file `sh.nanorc` to the nano text editor. Each of these files is specific to a suffix and subsequent language.

Referring back to the command hierarchy within this chapter, we can use a type to determine the location and type of file `hello.sh` is:

$ type hello1.sh  #To determine the type and path
$ type -a hello1.sh  #To print all commands found if the name is NOT unique
$ type -t hello1.sh  #To print the simple type of the command

These commands and output can be seen in the following screenshot:

```
pi@pilabs ~ $ type hello1.sh
hello1.sh is hashed (/home/pi/bin/hello1.sh)
pi@pilabs ~ $ type -a hello1.sh
hello1.sh is /home/pi/bin/hello1.sh
pi@pilabs ~ $ type -t hello1.sh
file
pi@pilabs ~ $ _
```

Hello Dolly!

It is possible that we might need a little more substance in the script than a simple fixed message. Static message content does have its place but we can make this script much more useful by building some flexibility.

In this chapter, we will look at positional parameters or arguments that we can supply to the script and in the next chapter we will see how we can make the script interactive and also prompt the user for input at runtime.
Running the script with arguments

We can run the script with arguments, after all it's a free world and Linux promotes your freedom to do what you want to do with the code. However, if the script does not make use of the arguments, then they will be silently ignored. The following code shows the script running with a single argument:

```
$ hello1.sh fred
```

The script will still run and will not produce an error. The output will not change either and will print `hello world`:

<table>
<thead>
<tr>
<th>Argument Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>The name of the script itself and is often used in usage statements.</td>
</tr>
<tr>
<td>$1</td>
<td>Positional argument, the first argument passed to the script.</td>
</tr>
<tr>
<td>${10}</td>
<td>Where two or more digits are needed to represent the argument position. Brace brackets are used to delimit the variable name from any other content. Single value digits are expected.</td>
</tr>
<tr>
<td>$#</td>
<td>Argument count is especially useful when we need to set the amount of arguments needed for correct script execution.</td>
</tr>
<tr>
<td>$*</td>
<td>Refers to all arguments.</td>
</tr>
</tbody>
</table>

For the script to make use of the argument, we can change the script content a little. Let's first copy the script, add in the execute permissions, and then edit the new `hello2.sh`:

```
$ cp $HOME/bin/hello1.sh $HOME/bin/hello2.sh
$ chmod +x $HOME/bin/hello2.sh
```

We need to edit the `hello2.sh` file to make use of the argument as it is passed at the command line. The following screenshot shows the simplest use of command line arguments allowing us now to have a custom message.

```
#!/bin/bash
echo "Hello $1"
exit 0
```
Run the script now, we can provide an argument as shown in the following:

$ hello2.sh fred

The output should now say **Hello fred**. If we do not provide an argument then the variable will be empty and will just print **Hello**. You can refer to the following screenshot to see the execution argument and output:

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ hello2.sh fred</td>
<td>Hello fred</td>
</tr>
</tbody>
</table>

If we adjust the script to use $*, all the arguments will print. We will see **Hello** and then a list of all the supplied arguments. If we edit the script and replace the `echo` line as follows:

```bash
echo "Hello $*"
```

Executing the script with the following arguments:

$ hello2.sh fred wilma  betty barney

Will result in the output shown in the following screenshot:

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ hello2.sh fred wilma betty barney</td>
<td>Hello fred wilma betty barney</td>
</tr>
</tbody>
</table>

If we want to print **Hello <name>**, each on separate lines, we will need to wait a little until we cover the looping structures. A for loop will work well to achieve this.

**The importance of correct quotes**

So far, we have used a simple double quoting mechanism to encase the strings that we want to use with `echo`.

In the first script, it does not matter if we use single or double quotes. The `echo "Hello World"` will be exactly the same as `echo 'Hello World'`.

However, this will not be the case in the second script so it is very important to understand the quoting mechanisms available in bash.
As we have seen, using the double quotes `echo "Hello $1"` will result in **Hello fred** or whatever the supplied value is. Whereas, if we use single quotes `echo 'Hello $1'` the printed output on the screen will be **Hello $1**, where we see the variable name and not its value.

The idea of the quotes is to protect the special character such as a space between the two words; both quotes protect the space from being interpreted. The space is normally read as a default field, separated by the shell. In other words, all characters are read by the shell as literals with no special meaning. This has the knock on effect of the $ symbol printing its literal format rather than allowing bash to expand its value. The bash shell is prevented from expanding the variable's value, as it is protected by the single quotes.

This is where the double quote comes to our rescue. The double quote will protect all the characters except the $, allowing bash to expand the stored value.

If we ever need to use a literal $ within the quoted string along with variables that need to be expanded; we can use double quotes but escape the desired $ with the backslash (\). For example, `echo "$USER earns \$4"` would print as **Fred earns $4** if the current user was Fred.

Try the following examples at the command line using all quoting mechanisms. Feel free to up your hourly rate as required:

```
$ echo "$USER earns $4"
$ echo '$USER earns $4'
$ echo "$USER earns \$4"
```

The output is shown in the following screenshot:

```
pi@pilabs ~ $ echo "$USER earns $4"
pi earns
pi@pilabs ~ $ echo '$USER earns $4'
$USER earns $4
pi@pilabs ~ $ echo "$USER earns \$4"
pi earns $4
pi@pilabs ~ $ 
```

**Printing the script name**

The $0 variable represents the script name and this is often used in usage statements. As we are not yet looking at conditional statements, we will have the script name printed above the displayed name.
Edit your script so that it reads as the following complete code block for $HOME/bin/hello2.sh:

```bash
#!/bin/bash
 echo "You are using $0"
 echo "Hello $*"
 exit 0
```

The output from the command is shown in the following screenshot:

```
pi@pilabs ~ $ hello2.sh fred
You are using /home/pi/bin/hello2.sh
Hello fred
pi@pilabs ~ $ ...
```

If we prefer not to print the path and only want the name of the script to show we can use the `basename` command, which extracts the name from the path. Adjusting the script so that the second line now reads is as follows:

```bash
echo "You are using $(basename $0)"
```

The $(...) syntax is used to evaluate the output of the inner command. We first run basename $0 and feed the result into an unnamed variable represented by the $.

The new output will appear as seen in the following screenshot:

```
pi@pilabs ~ $ hello2.sh fred
You are using hello2.sh
Hello fred
```

It is possible to achieve the same results using back quotes, this is less easy to read but we have mentioned this as you might have to understand and modify the scripts that have been written by others. The alternative to the $(...) syntax is shown in the following example:

```bash
echo "You are using 'basename $0'"
```

Please note that the characters used are back quotes and NOT single quotes. On UK and US keyboards, these are found in the top-left section next to the number 1 key.
Debugging your scripts

With the scripts as simple as we have seen so far, there is little that can go wrong or debug. As the script grows and decision paths are included with conditional statements, we may need to use some level of debugging to analyze the scripts progress better.

Bash provides two options for us, -v and -x.

If we want to look at the verbose output from our script and the detailed information about the way the script is evaluated line by line, we can use the -v option. This can be within the shebang but it is often easier to run the script directly with bash:

```
$ bash -v $HOME/bin/hello2.sh fred
```

This is especially useful in this example as we can see how each element of the embedded `basename` command is processed. The first step is removing the quotes and then the parentheses. Take a look at the following output:

```
pi@pilabs ~ $ bash -v $HOME/bin/hello2.sh fred
#!/bin/bash
echo "You are using $(basename $0)"
basename $0
basename $0
You are using hello2.sh
echo "Hello $*"
Hello fred
exit 0
pi@pilabs ~ $ _
```

More commonly used is the -x option, which displays the commands as they get executed. Its useful to know the decision branch that has been chosen by the script. The following shows this in use:

```
$ bash -x $HOME/bin/hello2.sh fred
```

We again see that the `basename` is evaluated first, but we do not see the more detailed steps involved in running that command. The screenshot that follows captures the command and output:

```
pi@pilabs ~ $ bash -x $HOME/bin/hello2.sh fred
++ basename /home/pi/bin/hello2.sh
  + echo 'You are using hello2.sh'
You are using hello2.sh
  + echo 'Hello fred'
Hello fred
  + exit 0
pi@pilabs ~ $ _
```
Summary
This marks the end of the chapter and I am sure that you might have found this useful. Especially for those making a start with bash scripting, this chapter must have built a firm foundation on which you can build your knowledge.

We began by ensuring that bash is secure and not susceptible to embedded functions shell-shock. With bash secured, we considered the execution hierarchy where aliases, functions, and so on are checked before the command; knowing this can help us plan a good naming structure and a path to locate the scripts.

Soon we were writing simple scripts with static content but we saw how easy it was to add flexibility using arguments. The exit code from the script can be read with the `$?` variable and we can create a command line list using `||` and `&&`, which depends on the success or failure of the preceding command in the list.

Finally, we closed the chapter by looking at debugging the script. Its not really required when the script is trivial, but it will be useful later when complexity is added.

In the next chapter, we will be creating interactive scripts that read the user's input during script execution.
Where to buy this book

You can buy Mastering Linux Shell Scripting from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.