Pro Bash Programming

Scripting the GNU/Linux Shell

Chris F.A. Johnson

Apress®

Pro Bash Programming: Scripting the GNU/Linux Shell

ISBN-13 (pbk): 978-1-4302-1997-2

ISBN-13 (electronic): 978-1-4302-1998-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Frank Pohlmann

Technical Reviewer: Ed Schaefer

Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Kylie Johnston

Copy Editor: Kim Wimpsett

Compositor: ContentWorks, Inc.

Indexer: Julie Grady

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 233 Spring Street, New York, NY 10013. E-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

Contents

About the Author

After almost 20 years in magazine and newspaper publishing, variously as writer, editor, graphic designer, and production manager, **Chris F.A. Johnson** now earns his living composing cryptic crossword puzzles, teaching chess, designing and coding web sites, and programming...and writing books about shell scripting. His first book, *Shell Scripting Recipes: A Problem-Solution Approach*, was published by Apress in 2005.

Introduced to Unix in 1990, Chris learned shell scripting because there was no C compiler on the system. His first major project was a menu-driven, user-extensible database system with a report generator. Constantly writing scripts for any and all purposes, his recent shell projects have included utilities for manipulating crossword puzzles and preparing chess resources for his students.

About the Technical Reviewer

Ed Schaefer is an ex-paratrooper, an ex-military intelligence officer, an ex-oil field service engineer, and a past contributing editor and columnist for *Sys Admin*. He's not a total has-been. He earned a BSEE from South Dakota School of Mines and an MBA from USD.

Currently, he fixes microstrategy and teradata problems—with an occasional foray into Linux—for a Fortune 50 company.

Introduction

Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design.

Brian Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984

The shell *is* a programming language. Don't let anyone tell you otherwise. The shell is not just glue that sticks bits together. The shell is a lot more than a tool that runs other tools. *The shell is a complete programming language!*

When a Linux user asked me about membership databases, I asked him what he really needed. He wanted to store names and addresses for a couple of hundred members and print mailing labels for each of them. I recommended using a text editor to store the information in a text file, and I provided a shell script to create the labels in PostScript. (The script, `ps-labels`, appeared in my first book, *Shell Scripting Recipes: A Problem-Solution Approach*.)

When the SWEN worm was dumping hundreds of megabytes of junk into my mailbox every few minutes, I wrote a shell script to filter them out on the mail server and download the remaining mail to my home computer. That script has been doing its job for several years.

I used to tell people that I did most of my programming in the shell but switched to C for anything that needed the extra speed. It has been several years since I have needed to use C, so I no longer mention it. I do everything in the shell.

A shell script is as much a program as anything written in C, Python, or any other language. Just because shell scripts are easier to write doesn't mean they should take a backseat to compiled programs or other scripting languages. I use the terms *script* and *program* interchangeably when referring to tasks written in the shell.

Why the Shell?

Some Linux users do all of their work in a GUI environment and never see a command line. Most, however, use the shell at least occasionally and know something about Unix commands. It's not a big step from there to saving oft-repeated commands in a script file. When they need to extend the capabilities of their system, the shell is the natural way to go.

The shell also has important advantages over other programming languages:

- It interfaces simply and seamlessly with the hundreds of Unix utilities.
- It automatically expands wildcards into a list of file names.
- Lists contained in a variable are automatically split into their constituent parts.

Just the Shell, Ma'am, Just the Shell

While most shell programs *do* call external utilities, a lot of programming can be done entirely in the shell. Many scripts call just one or two utilities for information that is used later in the script. Some scripts are little more than wrappers for other commands such as `awk`, `grep`, or `sed`.

This book is about programming in the shell itself. There's a sprinkling of the second type, where the script gets information (such as the current date and time) and then processes it. The third type gets barely more than a cursory nod.

A Brief History of sh

The Bourne shell was the first Unix shell in general use. It was much more limited than today's shells, so it *was* primarily a tool to run other tools. It had variables, loops, and conditional execution, but the real work was done almost entirely by external utilities.

The C shell, `csh`, added command history, arithmetic, and other features that made it popular as a command-line interpreter, but it was not suitable for more than trivial scripts.

The KornShell, developed by David Korn at AT&T Bell Labs, combined the Bourne shell syntax with features of the C shell. It was compatible with the Bourne shell, bringing important functionality into the shell itself and making script execution much faster. Until the year 2000, when it was opened up, `ksh` was proprietary, closed-source software.

The GNU Project, needing a free, open-source shell, introduced `bash`. Like all modern shells, `bash` is a POSIX shell. It also has many added enhancements.

Which Version of Bash?

This book is aimed at users of `bash-3` and later, but much of the book will work with `bash-2.05`. Features introduced in `bash-4.0`, are covered, but it is noted that they are newer additions. Wherever possible without loss of efficiency, the scripts in this book are written to run in any POSIX shell. It is often noted when a statement in a script is nonstandard or uses bash syntax. That means that it is not POSIX compliant. Most functions, however, will not run in a POSIX shell because they almost always use the `local` builtin command.

Some Linux distributions modify their versions of `bash`: Debian removes network socket capabilities. Mandriva removes the builtin `time` command. For this reason, I recommend compiling your own copy of `bash`. The source code is available from `http://tiswww.case.edu/php/chet/bash/bashtop.html`, and it compiles without trouble almost anywhere.

Who Will Benefit from This Book?

If you're an experienced shell programmer, this book will provide insight into the arcana of shell scripting, helping you write more, and more efficient, scripts.

If you have dabbled in shell scripting, I hope this book will encourage you to experiment further.

If you are new to shell scripting, this book will get you started and help you quickly become a proficient shell programmer.

No matter what your level of experience, this book will enable you to program tasks that aren't already dealt with on your system.

What's in the Book?

From writing your first program to using the mouse in your scripts, this book runs the gamut from simple to complex and from the obvious to the obscure. It covers every technique you will need to write efficient shell programs.

Chapter 1, Hello, World! Your First Shell Program, presents the traditional first program in any language. It prints "Hello, World!" The chapter discusses how to write the script, what to name it, and where to put it.

Chapter 2, Input, Output, and Throughput, demonstrates output using the `echo` and `printf` commands and introduces the `read` command for input. It also examines redirecting both input and output and connecting commands with pipelines.

Chapter 3, Looping and Branching, explains the looping statements, `for`, `while`, and `until`; the branching statement `if`; and the conditional operators `&&` and `||`.

Chapter 4, Command-Line Parsing and Expansion, describes how the shell parses a command line, from word splitting to parameter expansion.

Chapter 5, Variables and Parameters, covers all the possibilities of parameters and variables, from scalar variables to associative arrays and from default substitution to search and replace.

Chapter 6, Shell Functions, delves into the syntax of function definitions and defines a number of useful routines.

Chapter 7, String Manipulation, contains a number of functions for dicing and splicing strings.

Chapter 8, File Operations and Commands, uses more external commands than the rest of the book put together. That's because looping through a large file with the shell is painfully slow, and the Unix utilities are very efficient. This chapter also tells you when not to use those utilities.

Chapter 9, Reserved Words and Builtin Commands, looks at a number of commands that are built into the shell itself.

Chapter 10, Writing Bug-Free Scripts and Debugging the Rest, takes a buggy script and takes you step-by-step through fixing it, as well as showing you how to prevent bugs in the first place.

Chapter 11, Programming for the Command Line, is for those people who spend a lot of time at the command prompt. These programs and functions reduce the typing to a minimum.

Chapter 12, Runtime configuration, describes seven methods of altering a program's runtime behavior.

Chapter 13, Data Processing, deals with manipulating different types of data in the shell.

Chapter 14, Scripting the Screen, shows you how to write to the screen as if it were a canvas rather than a teletypewriter.

Chapter 15, Entry-Level Programming, presents techniques for getting input from a user, both using single keypresses and entering and editing a line of text. It concludes with routines for using the mouse in shell scripts.

The **appendix** lists all the variables used by `bash` with a description of each.

Downloading the Code

The source code for this book is available to readers as a gzipped tarball file at `www.apress.com` in the Downloads section of this book's home page. Please feel free to visit the Apress web site and download all the code there. You can also check for errata and find related titles from Apress.

Contacting the Author

The author maintains a web site at `http://cfajohnson.com/` and can be reached via e-mail at `shell@cfajohnson.com`.

CHAPTER 1

Hello, World! Your First Shell Program

A *shell script* is a file containing one or more commands that you would type on the command line. This chapter describes how to create such a file and make it executable. It also covers some other issues surrounding shell scripts, including what to name the files, where to put them, and how to run them.

I will begin with the first program traditionally demonstrated in every computer language: a program that prints "Hello, World!" in your terminal. It's a simple program, but it is enough to demonstrate a number of important concepts. The code itself is the simplest part of this chapter. Naming the file and deciding where to put it are not complicated tasks, but they are important.

For most of this chapter, you will be working in a terminal. it could be a virtual terminal, a terminal window, or even a dumb terminal. In your terminal, the shell will immediately execute any commands you type (after you press Enter, of course).

You should be in your home directory, which you can find in the variable `$HOME`:

```
echo $HOME
```

You can find the current directory with either the `pwd` command or the `PWD` variable:

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

If you are not in your home directory, you can get there by typing `cd` and pressing Enter at the shell prompt.

The Code

The code is nothing more than this:

```
echo Hello, World!
```

There are three words on this command line: the command itself and two arguments. The command, `echo`, prints its arguments separated by a single space and terminated with a newline.

The File

Before you turn that code into a script, you need to make two decisions: what you will call the file and where you will put it. The name should be unique (that is, it should not conflict with any other commands), and you should put it where the shell can find it.

The Naming of Scripts

Beginners often make the mistake of calling a trial script `test`. To see why that is bad, enter the following at the command prompt:

```
type test
```

The `type` command tells you what the shell will execute (and where it can be found if it is an external file) for any given command. In `bash`, `type -a test` will display all the commands that match the name `test`:

```
$ type test
test is a shell builtin
$ type -a test
test is a shell builtin
test is /usr/bin/test
```

As you can see, a command called `test` already exists; it is used to test file types and to compare values. If you call your script `test`, it will not be run when you type `test` at the shell prompt; the first command identified by `type` will be run instead. (I'll talk more about both `type` and `test` in later chapters.)

Typically, Unix command names are as short as possible. They are often the first two consonants of a descriptive word (for example, `mv` for **m**o**v**e or `ls` for **l**i**s**t) or the first letters of a descriptive phrase (for example , `ps` for **p**rocess **s**tatus or `sed` for **s**tream **ed**itor).

For this exercise, call the script `hw`. Many shell programmers add a suffix, such as `.sh`, to indicate that the program is a shell script. The script doesn't need it, and I use one only for programs that are being developed. My suffix is `-sh`, and when the program is finished, I remove it. A shell script becomes another command and doesn't need to be distinguished from any other type of command.

Selecting a Directory for the Script

When the shell is given the name of a command to execute, it looks for that name in the directories listed in the `PATH` variable. This variable contains a colon-separated list of directories that contain executable commands. This is a typical value for `$PATH`:

```
/bin:/usr/bin:/usr/local/bin:/usr/games
```

If your program is not in one of the `PATH` directories, you must give a pathname, either absolute or relative, for `bash` to find it. An *absolute* pathname gives the location from the root of the filesystem, such as `/home/chris/bin/hw`; a *relative* pathname is given in relation to the current working directory (which should currently be your home directory), as in `bin/hw`.

Commands are usually stored in directories named `bin`, and a user's personal programs are stored in a `bin` subdirectory in the `$HOME` directory. To create that directory, use this command:

```
mkdir bin
```

Now that it exists, it must be added to the `PATH` variable:

```
PATH=$PATH:$HOME/bin
```

For this change to be applied to every shell you open, add it to a file that the shell will *source* when it is invoked. This will be `.bash_profile`, `.bashrc`, or `.profile` depending on how `bash` is invoked. These files are sourced only for interactive shells, not for scripts.

Creating the File and Running the Script

Usually you would use a text editor to create your program, but for a simple script like this, it's not necessary to call up an editor. You can create the file from the command line using redirection:

```
echo echo Hello, World! > bin/hw
```

The greater-than sign (`>`) tells the shell to send the output of a command to the specified file, rather than to the terminal. You'll learn more about redirection in Chapter 2.

The program can now be run by calling it as an argument to the shell command:

```
bash bin/hw
```

That works, but it's not entirely satisfactory. You want to be able to type `hw`, without having to precede it with `bash`, and have the command executed. To do that, give the file execute permissions:

```
chmod +x bin/hw
```

Now the command can be run using just its name:

```
$ hw
Hello, World!
```

Choosing and Using a Text Editor

For many people, one of the most important pieces of computer software is a word processor. Although I am using one to write this book (OpenOffice.org Writer), it's not something I use often. The last time I used a word processor was four years ago when I wrote my previous book. A text editor, on the other hand, is an indispensable tool. I use one for writing e-mail, Usenet articles, shell scripts, PostScript programs, web pages, and more.

A text editor operates on plain-text files. It stores only the characters you type; it doesn't add any hidden formatting codes. If I type `A` and press Enter in a text editor and save it, the file will contain exactly two characters: *A* and a newline. A word-processor file containing the same text would be thousands of times larger. (With *abiword*, the file contains 2,526 bytes; the OpenOffice.org file contains 8,192 bytes.)

You can write scripts in any text editor, from the basic `e3` or `nano` to the full-featured `emacs` or `nedit`. The better text editors allow you to have more than one file open at a time. They make editing code easier with, for example, syntax highlighting, automatic indentation, autocompletion, spell checking, macros, search and replace, and undo. Ultimately, which editor you choose is a matter of personal preference. I use GNU `emacs` (see Figure 1-1).

■ **Note** In Windows text files, lines end with two characters: a *carriage return* (CR) and a *linefeed* (LF). On Unix systems, such as Linux, lines end with a single linefeed. If you write your programs in a Windows text editor, you must either save your files with Unix line endings or remove the carriage returns afterward.

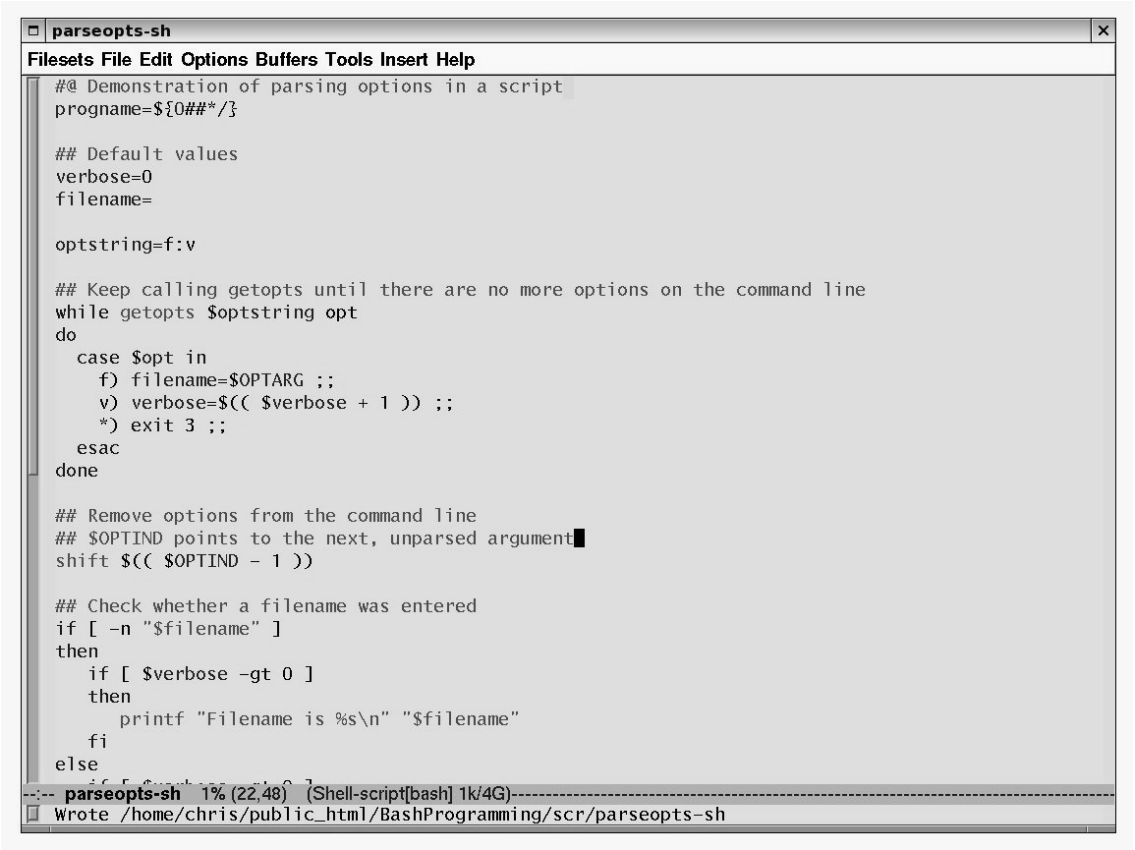

Figure 1-1. *Shell code in the GNU `emacs` text editor*

Building a Better "Hello, World!"

Earlier in the chapter you created a script using redirection. That script was, to say the least, minimalist. All programs, even a one-liner, require documentation. Information should include at least the author, the date, and a description of the command. Open the file `bin/hw` in your text editor, and add the information in Listing 1-1 using *comments*.

Listing 1-1. *hw*

```
#!/bin/bash
#: Title       : hw
#: Date        : 2008-11-26
#: Author      : "Chris F.A. Johnson" <shell@cfajohnson.com>
#: Version     : 1.0
#: Description : print Hello, World!
#: Options     : None

printf "%s\n" "Hello, World!"
```

Comments begin with an octothorpe, or *hash* (#), at the beginning of a *word* and continue until the end of the line. The shell ignores them. I often add a character after the hash to indicate the type of comment. I can then search the file for the type I want, ignoring other comments.

The first line is a special type of comment called a *shebang* or *hash-bang*. It tells the system which *interpreter* to use to execute the file. The characters `#!` must appear at the very beginning of the first line; in other words, they must be the first two bytes of the file for it to be recognized.

Summary

The following are the commands, concepts, and variables you learned in this chapter.

Commands

- `pwd`: Prints the name of the current working directory
- `cd`: Changes the shell's working directory
- `echo`: Prints arguments separated by a space and terminated by a newline
- `type`: Displays information about a command
- `mkdir`: Creates a new directory
- `chmod`: Modifies the permissions of a file
- `source: a.k.a. . (dot):` executes a script in the current shell environment
- `printf`: Prints the arguments as specified by a format string