THE EXPERT'S VOICE® IN OPEN SOURCE

SECOND EDITION

Shell Scripting Recipes

A Problem-Solution Approach

POSIX-COMPLIANT SCRIPTS FOR BASH, KSH, SH, AND MORE

Chris F. A. Johnson, Jayant Varma

Apress®

Shell Scripting Recipes

A Problem-Solution Approach

Second Edition

Chris F. A. Johnson
Jayant Varma

Apress®

Shell Scripting Recipes, Second Edition

ISBN-13 (pbk): 978-1-4842-0221-0

ISBN-13 (electronic): 978-1-4842-0220-3

Managing Director: Welmoed Spahr
Lead Editor: Louise Corrigan
Editorial Board: Steve Anglin, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Mark Powers
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 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 www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit 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 www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484202210. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

This book is dedicated to my parents,
who would have been quite proud to see this book.

—Jayant Varma

Contents at a Glance

Contents

About the Authors

Chris F. A. Johnson was introduced to Unix in 1990 and 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 report generator. Chris uses the shell as his primary, general-purpose programming language, and his projects have included a member database, menuing system, and POP3 mail filtering and retrieval. Chris is the coauthor of *Pro Bash Programming* (Apress, 2015). When not pushing shell scripting to the limit, he designs and codes web sites, teaches chess, and composes cryptic crosswords.

Jayant Varma is the founder of OZ Apps (`www.oz-apps.com`), a consulting, training, and development company providing IT solutions (specialization in mobile technology). He is an experienced developer with more than 20 years of industry experience spread across several countries. He is the author of a number of books on iOS development, including *Learn Lua for iOS Game Development* (Apress, 2012), *Xcode 6 Essentials* (Packt, 2015), *More iPhone Development with Swift* (Apress, 2015), and *More iPhone Development with Objective-C* (Apress, 2015). He has also been a university lecturer in Australia where he currently resides. He loves traveling and finds Europe to be his favorite destination.

Acknowledgments

I would like to thank the wonderful staff at Apress for the opportunity to update this book. Special thanks to Louise and Mark, who facilitated the quick turnaround on the book and getting it to print. Lastly, special thanks to my family for their support in getting this book completed.

—Jayant Varma

CHAPTER 1

The POSIX Shell and Command-Line Utilities

The POSIX shell is a descendant of the KornShell, which is a descendant of the Bourne shell. The basic syntax has remained the same, and Bourne shell scripts will usually run successfully in a POSIX shell. Not all of the KornShell features are included in POSIX (there are no arrays, for example), but the most important ones are. Those, which would be important to many developers, are string manipulation and arithmetic.

The scripts in this book make extensive use of features of the POSIX shell, and keep external commands to a minimum. This chapter presents an overview of the features of the POSIX shell and the external Unix commands used in this book, without going into great detail. Further information is available in the documentation for the shell and the various commands as well as on many web pages, a number of which are listed in the Appendix. We will also explain some of the idiosyncrasies used in the scripts, and present a library of functions that are used by many scripts.

Shell Commands

These descriptions are brief overviews of the built-in commands; for a complete description, see your shell's man page.

echo

The `echo` command prints its arguments separated by single spaces followed by a newline. If an unquoted variable contains characters present in `$IFS` (see "Parameters and Variables" later in this chapter), then the variable will be split on those characters:

```
$ list="a   b c d e f   g   h"
$ echo $list
a b c d e f g h
```

If the variable is quoted, all internal characters will be preserved:

```
$ echo "$list"
a   b c d e f  g  h
```

In the early days of Unix, two different versions of `echo` appeared. One version converted escape sequences, such as `\t` and `\n`, into the characters they represent in the C language; `\c` suppressed the newline, and discarded any further characters. The other used the `-n` option to suppress the trailing newline and did not convert escape sequences. The POSIX standard for `echo` says that "Implementations shall not support any options" and "If the first operand is `-n`, or if any of the operands contain a backslash ('\') character, the results are implementation-defined." In other words, you cannot rely on `echo`'s behavior being one or the other. It is best not to use `echo` unless you know exactly what `echo` is going to print, and you know that it will not contain any problem characters. The preferred command is `printf`.

printf

This command may be built into the shell itself, or it may be an external command. Like the C-language function on which it is based, `printf` takes a `format` operand that describes how the remaining arguments are to be printed, and any number of optional arguments. The `format` string may contain literal characters, escape sequences, and conversion specifiers. Escape sequences (the most common ones being `\n` for newline, `\t` for tab, and `\r` for carriage return) in `format` will be converted to their respective characters. Conversion specifiers, `%s`, `%b`, `%d`, `%x`, and `%o`, are replaced by the corresponding argument on the command line. Some implementations support other specifiers, but they are not used in this book. When there are more arguments than spec-ifiers, the `format` string is reused until all the arguments have been consumed.

The `%s` specifier interprets its argument as a string and prints it literally:

```
$ printf "%s\n" "qwer\ty" 1234+5678
qwer\ty
1234+5678
```

The `%b` specifier is like `%s`, but converts escape sequences in the argument:

```
$ printf "%b\n" "qwer\ty" "asdf\nghj"
qwer    y
asdf
ghj
```

The `%d`, `%x`, and `%o` specifiers print their arguments as decimal, hexadecimal, and octal numbers, respectively.

```
$ printf "%d %x %o\n" 15 15 15
15 f 17
```

The conversion specifiers may be preceded by flags for width specification, optionally preceded by a minus sign indicating that the conversion is to be printed flush left, instead of flush right, in the specified number of columns:

```
$ printf "%7d:\n%7s:\n%-7s:\n" 23 Cord Auburn
     23:
   Cord:
Auburn :
```

In a numeric field, a 0 before the width flag indicates padding with zeroes:

```
$ printf "%07d\n" 13
0000013
```

set

In the *Oxford English Dictionary*, the longest entry is for the word *set*—thirty-two pages in my Compact Edition. In the Unix shell, the `set` command is really three commands in one. Without any arguments, it prints the names and values of all shell variables (including functions). With one or more option arguments, it alters the shell's behavior. Any non-option arguments are placed in the positional parameters.

Only three options to `set` are used in this book:

- `-v`: Print shell input lines as they are read.
- `-x`: Print commands and their arguments as they are executed.
- `-f`: Disable file name generation (globbing).

Given this script, which is call `xx.sh`,

```
echo "Number of positional parameters: $#"
echo "First parameter: ${1:-EMPTY}"
shift $(( $# - 1 ))
echo "Last parameter: ${1:-EMPTY}"
```

its output is this:

```
$ xx.sh the quick brown fox
Number of positional parameters: 4
First parameter: the
Last parameter: fox
```

If `set -v` is added to the top of the script, and the standard output redirected to oblivion, the script itself is printed:

```
$ xx.sh the quick brown fox >/dev/null
echo "Number of positional parameters: $#"
echo "First parameter: ${1:-EMPTY}"
shift $(( $# - 1 ))
echo "Last parameter: ${1:-EMPTY}"
```

If `set -v` is replaced with `set -x`, variables and arithmetic expressions are replaced by their values when the lines are printed; this is a useful debugging tool:

```
$ xx.sh the quick brown fox >/dev/null
++ echo 'Number of positional parameters: 4'
++ echo 'First parameter: the'
++ shift 3
++ echo 'Last parameter: fox'
++ exit
```

To demonstrate the `set -f` option, and the use of `+` to reverse the operation, run the following script in an empty directory:

```
## Create a number of files using brace expansion (bash, ksh)
touch {a,b,c,d}${RANDOM}_{e,r,g,h}${RANDOM}

## Turn off filename expansion
set -f
printf "%-22s%-22s%-22s\n" * ; echo ## Display asterisk
printf "%-22s%-22s%-22s\n" *h* ; echo ## Display "*h*"

## Turn filename expansion back on
set +f
printf "%-22s%-22s%-22s\n" * ; echo ## Print all filenames
printf "%-22s%-22s%-22s\n" *h* ; echo ## Print filenames containing "h"
```

When the script is run, this is the output:

```
$ xx.sh
*

*h*

a12603_e6243           a28923_h23375          a29140_r28413
a5760_g7221            b17774_r4121           b18259_g11343
b18881_e10656          b660_h32228            c22841_r19358
c26906_h14133          c29993_g6498           c6576_e25837
d11453_h12972          d25162_e3276           d7984_r25591
d8972_g31551

a28923_h23375          b660_h32228            c26906_h14133
d11453_h12972
```

You can use `set` to split strings into pieces by changing the value of `$IFS`. For example, to split a date, which could be 2005-03-01 or 2003/09/29 or 2001.01.01, `$IFS` can be set to all the possible characters that could be used as separators. The shell will perform word splitting on any character contained in `$IFS`:

```
$ IFS=' -/.'
$ set 2005-03-01
$ printf "%s\n" "$@"
2005
03
01
```

When the value to be set is contained in a variable, a double dash should be used to ensure that the value of the variable is not taken to be an option:

```
$ var="-f -x -o"
$ set -- $var
```

shift

The leading positional parameters are removed, and the remaining parameters are moved up. By default, one parameter is removed, but an argument may specify more:

```
$ set 1 2 3 4 5 6 7 8
$ echo "$* ($#)"
1 2 3 4 5 6 7 8 (8)
$ shift
$ echo "$* ($#)"
2 3 4 5 6 7 8 (7)
$ shift 3
$ echo "$* ($#)"
5 6 7 8 (4)
```

■ **Tip** This is most useful when used in scripts to iterate through a number of parameters passed. Access the **$0** parameter and keep shifting till there are no more to go through. However shift removes the parameters, so be aware if you want to reuse the parameters.

Some shells will complain if the argument to shift is larger than the number of positional parameters.

type

The POSIX standard says `type` "shall indicate how each argument would be interpreted if used as a command name." Its return status may be used to determine whether a command is available:

```
if type stat > /dev/null 2>&1 ## discard the output
then
  stat "$file"
fi
```

If the command is an executable file, `type` prints the path to the file; otherwise, it prints the type of command that will be invoked: function, alias, or shell builtin. Its output is not standard across different shells, and therefore cannot be used reliably in a shell script.

The four arguments to `type` in the following example represent an executable file, a function, a nonexistent command, and an alias.

```
$ type ls pr1 greb ecoh
ls is hashed (/bin/ls)
pr1 is a function
pr1 ()
{
```

```
    case $1 in
        -w)
            pr_w=
        ;;
        *)
            pr_w=-.${COLUMNS:-80}
        ;;
    esac;
    printf "%${pr_w}s\n" "$@"
}
bash: type: greb: not found
ecoh is aliased to `echo'
```

Unlike most shells, `bash` will print the definition of a function.

getopts

The command getopts parses the positional parameters according to a string of acceptable options. If an option is followed by a colon, an argument is expected for that option, and will be stored in `$OPTARG`. This example accepts `-a`, `-b`, and `-c`, with `-b` expecting an argument:

```
while getopts ab:c opt
do

    case $opt in
        a) echo "Option -a found" ;;
        b) echo "Option -b found with argument $OPTARG" ;;
        c) echo "Option -c found" ;;
        *) echo "Invalid option: $opt"; exit 5 ;;
     esac
done
```

case

A workhorse among the shell's built-in commands, `case` allows multiple branches, and is the ideal tool, rather than `grep`, for determining whether a string contains a pattern or multiple patterns. The format is

```
case STRING in
    PATTERN [| PATTERN ...]) [list] ;;
    [PATTERN [| PATTERN ...]) [list] ;; ...]
esac
```

The `PATTERN` is a pathname expansion pattern, not a regular expression, and the `list` of commands following the first `PATTERN` that matches is executed. (See the "Patterns" section further on for an explanation of the two types of pattern matching.)

eval

The command `eval` causes the shell to evaluate the rest of the line, then execute the result. In other words, it makes two passes at the command line. For example, given the command:

```
eval "echo \${$#}"
```

The first pass will generate `echo ${4}` (assuming that there are 4 positional parameters). This will then print the value of the last positional parameter, `$4`.

local

The `local` command is used in functions; it takes one or more variables as arguments and makes those local to the function and its children. Though not part of the POSIX standard, it is built into many shells; `bash` and the `ash` family have it, and `pdksh` has it as a standard alias for `typeset` (which is also not included in POSIX). In KornShell 93 (generally referred to as `ksh93`), if a function is defined in the portable manner (as used throughout this book), there is no way to make a variable local to a function.

In this book, `local` is used only in the few scripts that are written specifically for `bash`, most often for setting `$IFS` without having to restore it to its original value:

```
local IFS=$NL
```

Parameters and Variables

Parameters are names used to represent information; there are three classes of parameters: Positional parameters are the command-line arguments, and are numbered beginning with $1; variables are parameters denoted by a name that contains only letters, numbers and underscores, and that begins with a letter or an underscore; and special parameters that are represented by non-alphanumeric characters.

Positional Parameters

Positional parameters are the command-line arguments passed to a script or a function, and are numbered beginning with `1`. Parameters greater then 9 must be enclosed in braces: `${12}`. This is to preserve compatibility with the Bourne shell, which could only access the first nine positional parameters; `$12` represents the contents of `$1`, followed by the number `2`. The positional parameters can be assigned new values, with the `set` command. (See the example under "Special Parameters.")

Special Parameters

The parameters `$*` and `$@` expand to all the positional parameters, and `#` represents the number of positional parameters. This function demonstrates the features of these parameters:

```
demo()
{
  printf "Number of parameters: %d\n" $#
  printf " The first parameter: %s\n" "$1"
  printf "The second parameter: %s\n" "$2"
  printf "\nAll the parameters, each on a separate line:\n"
  printf "\t%s\n" "$@"
```

```
  printf "\nAll the parameters, on one line:\n"
  printf "\t%s\n" "$*"
  printf "\nEach word in the parameters on its own line:\n"
  printf "\t%s\n" $*
}
```

Here, the demo function is run with three arguments:

```
$ demo The "quick brown" fox
Number of parameters: 3
 The first parameter: The
The second parameter: quick brown

All the parameters, each on a separate line:
        The
        quick brown
        fox

All the parameters, on one line:
        The quick brown fox

Each word in the parameters on its own line:
        The
        quick
        brown
        fox
```

The decimal exit code of the previous command executed (0 for success, non-zero for failure) is stored in $?:

```
$ true; echo $?
0
$ false; echo $?
1
```

The shell's current option flags are stored in $-; the shell's process ID is in $$; $! is the process ID of the most recently executed background command, and $0 is the name of the current shell or script:

```
$ sleep 4 &
[1] 12725
$ printf "PID: %d\nBackground command PID: %d\n" $$ $!
PID: 12532
Background command PID: 12725
$ printf "Currently executing %s with options: %s\n" "$0" "$-"
Currently executing bash with options: fhimBH
```

Shell Variables

These are the variables that are assigned values at the command line or in a script. The system or the shell itself also set a number of variables; those used in this book are

- `$HOME`: The path name of user's home directory (e.g., `/home/chris`).
- `$IFS`: A list of characters used as internal field separators for word splitting by the shell. The default characters are space, tab, and newline. Strings of characters can be broken up by changing the value of `$IFS`:

```
$ IFS=-; date=2005-04-11; printf "%s\n" $date
2005
04
11
```

- `$PATH`: This colon-separated list of directories tells the shell which directories to search for a command. To execute a command in other directories, including the current working directory, an explicit path must be given (`/home/chris/demo_script` or `./demo_script`, not just `demo_script`).
- `$PWD`: This is set by the shell to the pathname of the current working directory:

```
$ cd $HOME && echo $PWD
/home/chris
$ cd "$puzzles" && echo $PWD
/data/cryptics
```

standard-vars—A Collection of Useful Variables

My `standard-vars` file begins with these lines:

```
NL='
'
CR='
'
TAB=' '
```

You might be able to guess that these three variables represent newline, carriage return, and tab, but it's not clear, and cannot be cut and pasted from a web site or newsgroup posting. Once those variables are successfully assigned, however, they can be used, without ambiguity, to represent those characters. The `standard-vars` file is read by the shell and executed in the current environment (known as *sourcing*, it is described later in the chapter) in most of my shell scripts, usually via `standard-funcs`, which appears later in this chapter.

Create the file with the following script, then add other variables as we found them useful:

```
printf "%b\n" \
     "NL=\"\n\"" \
     "CR=\"\r\"" \
     "TAB=\"\t\"" \
     "ESC=\"\e\"" \
     "SPC=\"\040\" \
     "export NL CR TAB ESC SPC" > $HOME/scripts/standard-vars-sh
```

The `-sh` extension is part of the system used for working on scripts without contaminating their production versions. It is explained in Chapter 20.