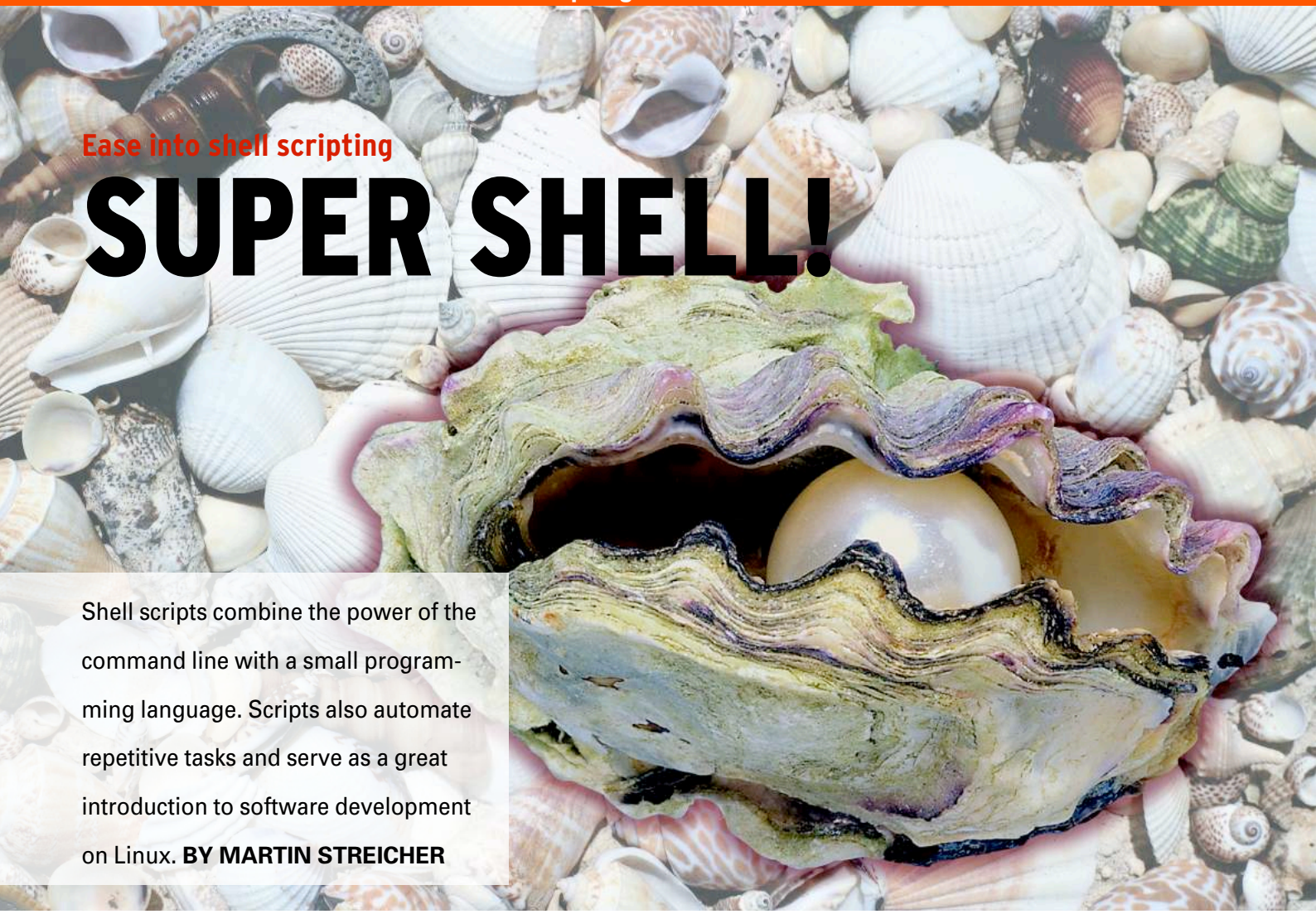

Ease into shell scripting

SUPER SHELL!

Shell scripts combine the power of the command line with a small programming language. Scripts also automate repetitive tasks and serve as a great introduction to software development on Linux. **BY MARTIN STREICHER**

One of the fundamental tenets of Linux (and its progenitor, Unix) is "Make each program do one thing well." This notion encourages simplicity and efficiency and is embodied in the raft of small, specialized utilities that can be found on a Linux system. For example, the sole function of *ls* is to list files and directories, while *df* lists only mounted volumes. Each utility has its own purpose, and there is little or no overlap among the system's large collection of available tools. (Of course, you often have a choice of many tools with the same purpose. For example, you can use *vi*, *kate*, or *pico*, among others, to edit text files.)

Another principle tenet is "Everything is a file." An application is a file; a directory is a file that catalogs other files; each device, too, is a file.

The commonality? Every file, no matter its semantics, can be opened, read, written to, and closed. Thus, in Linux, input can come from a terminal or an Internet connection, and output can be sent to a printer or to another running application.

Taken together, these two axioms allow two or more small applications to be combined ad hoc into a greater whole. The pipe (|) operator in the shell is the most visible implementation. It chains commands, making the output of one command the input of a subsequent command. For example, the *find* command can search a directory hierarchy for files that match certain criteria. The command:

```
$ find ~ -name '*homework.txt' -print
```

prints the names of all files in your home directory that begin with any string (the * is a wildcard for no or any number of characters) and end with "homework.txt". Thus, *homework.txt* would match, as would *math-homework.txt*. Separately. the *egrep* command searches the file content for a pattern. The command:

```
$ egrep -e '(math|science)' stuff
```

would search the file *stuff* for occurrences of the pattern "math" or the pattern "science".

If you combine the two previous commands with a pipe so that the output of *find* becomes the input of *egrep*, like so:

```
$ find ~ -name '*homework.txt' ➥
-print  | egrep -e '(math|science)'
```

you can find all homework assignments with "math" or "science" in the file's name. Assuming you had the files *home-*

Listing 1: Bash shell script to find specific files

```
01 #! /bin/bash
02
03 # Locate my math and science
   homework
04 find ~ -name '*homework.txt' -print
   | egrep -e '(math|science)'
```

Listing 2: A shell script that accepts one argument

```
01 #! /bin/bash
02
03 # Locate my math and science
   homework
04 find ~ -type f -name '*.txt' -print
   | egrep -e $1
```

work.txt, math-homework.txt, science-homework.txt, and *chem-homework.txt*, your new command would produce two results:

```
$ find ~ -name '*homework.txt' ➥
-print  | egrep -e '(math|science)'
./math-homework.txt
./science-homework.txt
```

This demonstration is just a simple example, but it shows how easy it is to instantly create a new tool with just a few keystrokes. Given the more than 1,000 utilities on a typical Linux system, the possibilities are endless.

In fact, many command combinations are perennially useful. You can capture a short command in an *alias*, a kind of shorthand, and save it for use, session after session. If you add the line:

```
alias fw="find ~ -name ➥
'*homework.txt' -print  | ➥
egrep -e '(math|science)'"
```

to your shell startup file, *~/.bashrc*, and type *source ~/.bashrc* or start a new shell, you can run the entire command with the abbreviation *fw*.

```
$ fw
/home/strike/class/math-homework.txt
/home/strike/class/science-homework.txt
```

You can also capture a command combination in a shell script. A shell script, which can be one or any number of lines, is an application composed of shell commands.

You can execute a script just as you would any system utility. It can leverage all the common shell operators – globbing, pipes, subshells, etc. – and other programming language features not typically used on the command line. Shell scripts are also portable. Because a script is nothing more than a text file, sharing it with others and propagating it to other systems is a snap. Expert users, system administrators, and developers accumulate shell scripts to accomplish a wide variety of tasks, and so can you.

Shell Script Anatomy

Let's look at shell scripting with the Bash shell, the most common Linux shell available. Almost every shell is scriptable, although syntax and features vary. Listing 1 shows a short shell script that captures the *find/egrep* combination shown previously.

To use the script, launch your favorite text editor and create a file named *findwork.sh* in your home directory with the lines shown in Listing 1. Do not put any leading spaces on the first line. Save, quit the editor, and run the following command:

```
$ chmod +x findwork.sh
```

chmod +x makes the file executable, so you can launch it just like any other command-line utility.

```
$ ./findwork.sh
/home/strike/class/math-homework.txt
/home/strike/class/science-homework.txt
```

Listing 3: Analyze file types based on file extension

```
01 #! /bin/bash
02
03 if [ -z $1 ]; then
04   echo "filetyper: No file
   specified"
05   exit 1
06 fi
07
08 for filename in "$@"; do
09   echo -n "$filename is a"
10
11   case $filename in
12     *.sh          )
13                     if [ -x $filename
   ]; then
14                       echo -n "n
   executable"
15                     fi
16                     echo -n "
   script";;
17
18     *.txt         ) echo -n "
   text";;
19
20     *.png | *.jpg ) echo -n "
   image";;
21
22     *             ) echo -n "n
   unknown";;
23   esac
24
25   echo " file"
26 done
27
28 exit 0
```

The first line of the script is special and is required for all executable scripts. The first two characters, *#!* (referred to as "pound-bang") identify the file as an executable script. The rest of the line, */bin/bash*, specifies what to run to interpret the script. Here, the script is a Bash shell script, hence */bin/bash*. (If you were writing a script in the Perl language, the first line of that file would be *#! /usr/bin/perl*.)

The second and subsequent lines of the file are the actual code. Blank lines are ignored, as is any line that begins with a *#*, which is treated as a comment. On line 4, as it does at the command line, Bash interprets and runs the command. Since the output of *egrep* is not piped to a later command, the output goes to standard output, your screen.

Typing *./findwork.sh* is terse and so much faster than retyping the entire command line, but it's inflexible.

If you need to search for math, science, and chemistry homework (*egrep -e '(math|science|chem)'*), you can either edit the script and make a temporary change, re-type the command on the command line, or. preferably, change the script to take an "argument". When you type a command such as *ls -l thisfile thatfile*, the phrases after the command

Listing 4: Output of Listing 3

```
01 $ ls -F
02 filetyper.sh           lab_results/
   parrot.png
03 findany.sh*
   math-12-homework.txt
   science-homework.txt
04 findwork.sh*
   math-homework.txt
05 homework.txt           monkey.jpg
06
07 $ ./filetyper.sh *
08 filetyper.sh is a script file
09 findany.sh is an executable script
   file
10 findwork.sh is an executable script
   file
11 homework.txt is a text file
12 lab_results is an unknown file
13 math-12-homework.txt is a text file
14 math-homework.txt is a text file
15 monkey.jpg is a image file
16 parrot.png is a image file
17 science-homework.txt is a text file
```

name are the arguments. Some arguments are interpreted as options or switches to affect the behavior of the command, such as *-l*, which enables a long format output. Any arguments that remain are acted upon. (Think of the command name as the verb and the arguments as the subjects.) Your shell scripts can accept arguments, too.

Listing 2 is a variant of Listing 1 that accepts a string as the filename pattern. The *find* command is slightly different in that it looks only for plain files (*-type f*) whose names end with ".txt". The special variable *$1* is replaced with the first argument from the command line, *$2* with the second, and so on. (Keep in mind that the command name does not count as an argument.)

Save this new script into *findany.sh*, make it executable with *chmod* as before, and invoke it with a pattern. Assuming you also had a file named chem-homework.txt, the command *./findany.sh '(math|chem|science)'* would yield three results:

```
$ ./findany.sh '(math|chem|science)'
/home/strike/class/math-homework.txt
/home/strike/class/science-homework.txt
/home/strike/class/chem-homework.txt
```

The pattern appears in single quotes because the characters *(*, *)*, and | are shell operators. The single quotes, which are also shell operators, disable interpretation, making the phrase one long string. Omitting the single quotes yields *bash: syntax error near unexpected token `math'*. "One-liners," like the scripts above, can save vast amounts of time.

Listing 5: Replacement code for Listing 3 to deduce that a file name is a directory

```
01 ...
02     *            ) if [ -d $filename
   ]; then
03                     echo "
   directory"
04                     continue
05                   else
06                     echo -n "n
   unknown"
07                   fi;;
08
09 ...
```

You might be surprised at how many great one-liners exist – indeed, writing them is something of an art form. However, to realize any significant automation, a shell script must be able to query its environment, make decisions, maintain state, and even interact with the user. In other words, the shell script must be a program.

Variables, Prompts, and Control Structures

Like C, Ruby, or SQL, Bash provides a fairly rich programming language of its own. You can set and test variables; you can branch, loop, and call subroutines; and you can mix in shell operators and Linux utilities. A full treatise on Bash isn't possible here. For an in-depth look at Bash scripting, consult the book *Learning the Bash Shell* [1].

The shell script in Listing 3 shows a number of control structures and coding paradigms that appear frequently in scripts. The script accepts a list of files and deduces the type of each file based on the file's extension (the portion after the dot). Let's step through Listing 3:

- The *if* statement at lines 3-6 determines whether or not the first argument is empty; if it is empty, implying no arguments, the script exits with a helpful message. The *-z* is a special *primary* operand used in conditions. It asks, "Is the string empty?" There are other primaries, too. Line 13 uses *-x* to test whether the named file both exists and is executable. *-d somefile* and *-w somefile* (both not shown) test whether a named file exists and is a directory, or if the named file is writeable, respectively.
- The *exit* in Line 5 terminates the script immediately. The argument, *1*, signals that the script ended with an error, which the command line and other scripts can detect and recover from if necessary. If at least one argument is provided, the script proceeds and exits with code *0*, indicating success.
- Line 8 is a loop. The extent of the loop begins with *do* and ends with *done*. This loop iterates over the collection of command-line arguments (abbreviated *$@*), setting the variable *filename* to one argument at a time. You can expand, or extract, the value of filename using the expression *$filename*. While *$1*, *$2*, and so on, are useful in one-liners, scripts that process many arguments work more like this.
- Line 9 prints a string without a trailing newline (the *-n* option). A double-quoted string interpolates variables; a single quoted string does not. Hence, this line prints something like "chem-homework.txt is a". (If this were written '*$filename is a*', the output would be the literal "$filename is a".)
- Line 11 is a *switch* statement. It's essentially a series of tests but is more compact than writing many if state-

Listing 6: Bash offers arrays

```
01 onestring=`ls -1`
02 manystrings=( `ls -1` )
03
04 echo $onestring
05 echo ${onestring[0]}
06 echo ${onestring[1]}
07
08 echo ${manystrings[0]}
09 echo ${manystrings[1]}
10 echo ${manystrings[2]}
```

Listing 7: You can embed text using a "here" document

```
01 TEMP=/tmp/usagenote.$$
02
03 users=(ethan joe sue)
04
05 for user in "${users[@]}"; do
06   usage=`du -h -c -d 1 /
   Users/$user`
07   cat > $TEMP <<-END
08     Dear $user:
09
10     Here is your daily disk usage
   report.
11
12     $usage
13
14     Please remove any unnecessary
   files.
15
16     The Management
17 END
18
19    mail -s "Daily usage report"
   $user < $TEMP
20 done
21
22 rm $TEMP
23
24 exit 0
```

ments. It tests the value of *$filename* against one or more values. Here, the values are shell glob patterns. If the value of filename matches **.sh* (any filename that ends with ".sh"), the statements associated with that value execute. If the value matches another pattern, that code runs. The double-semicolons (;;) are not typos. Each is part of the syntax of the switch statement: each ends a block of statements.

- Line 25 emits a last string and a trailing newline, and the loop begins anew if additional arguments remain. Otherwise, the script will terminate normally.

Save the script code in a file named *filetyper.sh*, make it executable, and run the new command with a list of file names. Running this code on a set of miscellaneous files might produce something like the output shown in Listing 4.

The output looks correct, except the directory *lab_results* is labeled an unknown type. That's easily remedied using the primary *-d*, as shown in Listing 5. This replacement code does the trick.

One addition is *continue*; it ends the current cycle of the loop and skips any remaining statements. You can also use *break* to immediately terminate the loop's iterations and continue with the rest of the script. Bash provides a *for*, *while*, and *until* loop; *continue* and *break* work identically in all three.

The Backtick and Arrays

Because shell scripts are most often used to automate what you might otherwise do at the command line, there are many techniques to capture information from the system to help make decisions.

You can use the backtick (`) operator to capture the output of a command in a variable. For example, if a directory has the files *apple, banana*, and *coconut*, the following code:

```
files=`ls -1`
echo $files
```

would emit the long string "apple banana coconut". You may not add whitespace around the equal sign in an assignment statement. If you do, it will change the semantics of the statement entirely. In some instances, compressing the output into one string is appropriate. However, if you wanted a true list of file names, you can use a variation in the assignment statement. Listing 6 shows the difference. Listing 6 produces this output:

```
files scripting.html src
files scripting.html src

files
scripting.html
src
```

onestring is a single element, so lines 4 and 5 produce the same result. Line 6 prints nothing because a second element does not exist (Bash arrays are zero-based, so the second element is at index 1). On the other hand, line 2 uses the assignment, *()*, operators to split a whitespace-delimited list into an array of three separate elements. As in line 8 of Listing 3, you can iterate over an array like so:

```
for string in "${manystrings[@]}"; do
echo $string
done
```

Embedded Data

Another use of automated scripts is to police system resources and alert users to emerging or urgent problems. Usually, an alert script contains a message. You can embed the message in your script and actually interpolate variables at the same time to customize the message to a particular user. Listing 7 shows an example.

In Listing 7, the lines between *<<-END* and *END* form the embedded text, or what's called a "here" document (as in "here it is"). The use of *END* as a start and end marker is arbitrary; you could use MONKEY or STOP, too. However, your start and end markers must match, and you must put the end marker at the start of a new line. If you change the script to name yourself, you should see output like Listing 8. Here are some other curiosities found in Listing 7:

- *$$* is a special variable that contains the process ID of the running shell. Because a process ID is unique, you can use it as a suffix to set aside a unique temporary file. The purpose of line 1 is to capture the unique name.
- Much like double-quotes, variables are interpolated within backticks (as in line 6).
- Line 7 uses *cat* and the overwrite redirection operator, >, to emit the interpolated here document to the temporary file. Because > recreates the file each time, there is no need to purge the file each time through the loop.
- All of the variables in the here document are expanded in place. The operator <<- removes all *leading tabs but not spaces* from the lines of the here document. If you changed <<- to <<, the here document will be used verbatim. If you do not want variables interpolated, place single quotes around the start market, as in '*END*'.
- The *-s* option of *mail* supplies the subject line. The body of the message is equivilant to the contents found in the temporary file.

INFO

[1] Newham, Cameron and Bill Rosenblatt. *Learning the Bash Shell*, 3rd Edition. O'Reilly Media, 2005.

Another source of data for a script is you or another user. Shell scripts can prompt a user for feedback, collect an answer, and continue.

User Input

To read data in Bash, use *read*. Its arguments are variable names. Each line of input is split by whitespace, and each word is assigned to a variable in sequence. If there are more words than variables, the last variable gets all remaining words. If there are no variables, the entire line of input is assigned to the special variable *REPLY*.

Listing 9 shows the permutations. Listing 10 shows you the results.If you want to split a line into words, such as to convert REPLY into an array, you can use an assignment statement. *words = ($REPLY)* does the trick.

Bash and other shells offer many more features than can be shown in one article. If you're curious to read more and see lots of examples, just search the Web for "bash script".

A Lot More...

Shell scripts are a great way to make yourself more productive. If you type a command or perform a task more than two or three times, your command or task is a good candidate for an alias, a one-liner, or a larger script.

Shell scripts also provide a great introduction to developing software on Linux. Indeed, many developers are first exposed to system programming right at the command line. ■

Listing 8: The output of Listing 7 for one user

```
01 Dear ethan:
02
03 Here is your daily disk usage
   report.
04
05 4.0K    /Users/ethan/.cups
06 8.0K    /Users/ethan/Desktop
07  23M    /Users/ethan/Documents
08 488K    /Users/ethan/Downloads
09  63M    /Users/ethan/Library
10 1.8G    /Users/ethan/Movies
11 120K    /Users/ethan/Music
12 228M    /Users/ethan/Pictures
13   0B    /Users/ethan/Public
14  40K    /Users/ethan/Sites
15 2.7G    /Users/ethan
16 2.7G    total
17
18 Please remove any unnecessary
   files.
19
20 The Management
```

THE AUTHOR

Martin Streicher is a freelance developer and author. He holds an advanced degree in computer science from Purdue University and has worked on software from a Unix assembler to the award-winning "You Don't Know Jack" CD-ROM game. You can reach Martin at: *martin.streicher@gmail.com*.

Listing 9: Techniques to collect user input

```
01 echo -n "Provide any list: " >&2
02 read
03 echo $REPLY
04
05 echo -n "Provide another list: "
   >&2
06 read list
07 echo $list
08
09 echo -n "Provide a list of three
   items: " >&2
10 read first second third
11 echo "first: $first, second:
   $second, third: $third"
12
13 echo -n "Provide a list of at least
   four items: " >&2
14 read first second third
15 echo "first: $first, second:
   $second, third: $third"
```

Listing 10: Results of user input

```
01 Provide any list: dog cat fish
02 dog cat fish
03
04 Provide another list: monkey
   giraffe llama
05 monkey giraffe llama
06
07 Provide a list of three items: toad
   lizard snake
08 first: toad, second: lizard, third:
   snake
09
10 Provide a list of at least four
   items: one two three four
11 first: one, second: two, third:
   three four
```