Getting started with Bash scripting

CUSTOM SCRIPT

A few scripting tricks will help you save time by automating common tasks. **BY ÆLEEN FRISCH**

Shell scripts are a lazy person's best friend. That may sound strange, because writing a shell script presumably takes work, but it's true. Writing a shell script to perform a repetitive task requires some time up front, but once the script is finished, using it frees up the time the task used to take. In this article, I will introduce you to writing shell scripts with Bash. I'll describe Bash scripting in the context of several common tasks. Because this is an introductory discussion, some nuances are glossed over or ignored, but I will provide plenty of information for you to get started on your own scripts.

Hello, Bash

In its simplest form, a shell script is just a file containing a list of commands to execute. For example, here is a script that a user created to avoid having to type a long *tar* command every time she wanted to back up all her picture files:

```
#!/bin/bash

tar cvzf /save/pix.tgz ↩
   /home/chavez/pix  /graphics/rdc ↩
   /new/pix/rachel
```

The script begins with a line that identifies the file as a script. The characters *#!* are pronounced "shbang," and the full path to the shell follows: In this case, the Bash executable. The remainder of the script is the *tar* command to run.

One more step is necessary before this script can actually be used. The user must set the executable file permission on the file so that the shell will know that it is a runnable script. If the script file is named *mytar*, the following *chmod* command does the trick (assuming the file is located in the current directory):

```
$ chmod u+x mytar
$ ./mytar
```

The second command runs the script, and many messages from tar will follow.

So far, the user has reduced the work required to create the tar archive from typing 75 characters to typing eight characters. However, you could make the script slightly more general – and potentially more useful – by putting the items to be saved on the command line:

```
$ ./mytar /home/chavez ↩
   /new/pix/rachel /jobs/proj5
```

This command backs up a different set of files. The modified script is shown in Listing 1, and it illustrates several new features:

- The *tar* command now uses I/O redirection to suppress non-error output.
- The *tar* command is conditionally executed. It is placed inside an *if* statement. If the test condition specified in the square brackets is true, then commands following are executed; otherwise, they are skipped.
- The if condition here determines whether the number of argument specified to the script, indicated by the *$#* construct, is greater than *0*. If so, then the user lists some items to back up. If not, then the script was run without arguments and there is nothing to do, so the *tar* command won't run.
- The script's command-line arguments are placed into the *tar* command via the *$@* construct, which expands to the argument list. In this example, the command will become:

```
tar czf /save/mystuff.tgz ↩
   /home/chavez /new/pix/rachel ↩
   /jobs/proj5  >/dev/null
```

Placing command-line arguments into the *tar* command allows the script to back up the necessary files.

Input File

The next incarnation of the script changes how it is run slightly (Listing 2). Now, the first command argument is assumed to be the name of a file containing a list of directories to back up. Any additional arguments are treated as literal items to be backed up.

DIRS and *OUTFILE* are variables used within the script. I'll use the convention of uppercase variable names to make them easy to identify, but this is not required. The first command in the script places the contents of the file specified

Listing 1: Modified Backup Script

```
#!/bin/bash

if [ $# -gt 0 ]; then     # Make
sure there is at least one argument
   tar czf /save/mystuff.tgz $@ >
   /dev/null
fi
```

Listing 2: Specifying an Input File

```
#!/bin/bash

DIRS="`cat $1`"                # DIRS = contents of file in 1st argument
shift                          # remove 1st argument from the list
OUTFILE="$( date +%y%m%d )"    # create a date-based archive name
tar czf /tmp/$OUTFILE.tgz $DIRS $@ >/dev/null
```

as the script's first argument into *DIRS*. This is accomplished by capturing the *cat* command output via back quotes.

Back quotes run whatever command is inside them and then place that command's output within the outer command, which then runs. Here, the *cat* command will display the contents of the file specified as the script's first argument – the directory list – and place it within the double quotes in the assignment statement, creating the variable *DIRS*. Note that line breaks in the directory list file do not matter.

Once I've read that file, I am done with the first argument, so I remove it from the argument list with the *shift* command. The new argument list contains any additional directories that were specified on the command line, and *$@* will again expand to the modified argument list. This mechanism allows the script user to create a list of standard items for backup once, but also to add additional items when needed.

The third command defines the variable *OUTFILE* using the output of the *date* command. The syntax here is a variant form of back quoting: *`command`* is equivalent to *$(command)*. This type of operation is known as command substitution. The final command runs *tar*, specifying the items from the first argument file and any additional arguments as the items to be backed up. Note that when you want to use a variable within another command, you precede its name by a dollar sign: *$DIRS*.

Adding Checks

Listing 2 is not as careful as the previous example in checking that its arguments are reasonable. Listing 3 shows the beginning of a more sophisticated script that restores this checking and provides more flexibility. This version uses the getopts feature built into Bash to process arguments quickly.

The first two commands assign values to the *DEST* and *PREFIX* variables, which specify the directory where the tar archive should be written and the archive name prefix (to be followed by a date-based string). The rest of this part of the script is structured as a *while* loop:

```
while condition-cmd;
   commands
done
```

The loop continues as long as the condition is true and exits once it becomes false. Here, the condition is *getopts "f:bn:d: " OPT*. Conditional expressions are enclosed in square brackets (as seen in the preceding and following *if* statements), but full commands are not (technically, the square brackets invoke the test command). Commands are true while they are returning output, and false when their output is exhausted.

The getopts tool returns each command-line option, along with any arguments. The option letter is placed into the variable specified as *getopts'* second argument – here *OPT* – and any argument is placed into *OPTARG*. *getopts'* first argument is a string that lists valid option letters (it is case sensitive); letters followed by colons require an argument – in this case, *f*, *n*, and *d*. When specified on the command line, option letters are followed by a hyphen.

The command inside the *while* loop is a *case* statement. This statement type checks the value of the item specified as its argument – here, the variable *OPT* set by getopts – against the series of patterns specified below. Each pattern is a string, possibly containing wildcards,

Listing 3: Restoring Checking

```
#!/bin/bash

DEST="/save"                          # Set default archive location & file prefix
PREFIX="backup"

while getopts "f:bn:d:" OPT; do       # Examine command line arguments
   case $OPT in                       # Specify valid matching patterns
      n) PREFIX=$OPTARG ;;            # -n <prefix>
      b) ZIP="j"; EXT="tbz" ;;        # -b = use bzip2 not gzip
      f) DIRS=$OPTARG  ;;             # -f <dir-list-file>
      d) if [ "${OPTARG:0:1}" = "/" ]   # -d <archive-dir>
         then
            DEST=$OPTARG
         else
            echo "Destination directory must begin with /."
            exit 1                    # end script with error status
         fi
         ;;
      :) echo "You need to give an argument for option -$OPTARG."
         exit 1
         ;;
      *) echo "Invalid argument: -$OPTARG."
         exit 1
         ;;
   esac
done
```

Listing 4: Restoring Checking (continued)

```
if [ -z $DIRS ]; then       # Make sure we have a valid item list file
    echo "The -f list-file option is required."
    exit 1
elif [ ! -r $DIRS ]; then
    echo "Cannot find or read file $DIRS."
    exit 1
fi

DAT="$( /bin/date +%d%m%g )"
/bin/tar -${ZIP-z} -c -f /$DEST/${PREFIX}_$DAT.${EXT-tgz} `cat $DIRS` >
    /dev/null
```

terminated by a closing parenthesis. Ordering is important because the first matching pattern wins.

In this example, the patterns are the valid option letters, a colon, and an asterisk wildcard matching anything other than the specified patterns (i.e., other than *n*, *b*, *f*, *d*, or *:*). The commands to process the various options differ, and each section ends with two semicolons. From the commands, you can see that *-n* specifies the archive name prefix (overriding the default set in the script's second command), *-b says* to use bzip2 rather than gzip for compression (as shown later), *-f* specifies the file containing the list of items to be backed up, and *-d* specifies the destination directory for the archive file (which defaults to */save* as before via the first command).

The destination directory is checked to make sure that it is an absolute pathname. The construct *${OPTARG:0:1}* deserves special attention. The most general form of $ substitution places curly braces around the item being dereferenced: *$1* can be written as *${1}*, and *$CAT* as *${CAT}*. This syntax is useful. It allows you to access positional parameters beyond the ninth; *${11}* specifies the script's 11th parameter, for example, but *$11* expands to the script's first argument followed by *1*: *${1}1*. The syntax also enables variables to be isolated from surrounding text: If the value of *ANIMAL* is *cat*, then *${ANIMAL}2* expands to *cat2*, whereas *$ANIMAL2* refers to the value of the variable *ANIMAL2*, which is probably undefined. Note that periods are not interpreted as part of variable names (as shown later).

The *:0:1* following the variable name extracts the substring from *OPTARG* beginning at the first position (character numbering starts at 0) and continuing for *1* character: in other words, its first character. The *if* command checks whether this character is a forward slash, displaying an error message if it is not and exiting the script with a status value of *1*, indicating an error termination (*0* is the status code for success).

When an option requiring an argument doesn't have one, getopts sets the variable *OPT* to a colon and the corresponding option string is put into *OPTARGS*. The penultimate section of the *case* statement handles these errors. The final section handles any invalid options encountered. If this happens, getopts sets its variable to a question mark and places the unknown option into *OPTARGS*; the wildcard pattern will match and handle things if this event occurs.

This argument handling code is not bulletproof. Some invalid option combinations are not detected until later in the script (e.g., *-f -n: -f*'s argument is missing, so *-n* is misinterpreted as such).

The remainder of the script started in Listing 3 is shown in Listing 4.

The *if* statement checks for two possible problems with the file containing the directory list. The first test checks whether the variable *DIRS* is undefined (has zero length), exiting with an error message if this is the case. The second test, following *elif* (for "else-if") makes sure the specified file exists and is readable. If not (the exclamation point in the expression serves as a logical NOT), the script gives an error message and exits.

The final two commands create the date-based part of the archive name and run the *tar* command. The *tar* command uses some conditional variable dereferencing – for example, *${EXT-tgz}*. The hyphen following the variable name says to use the following string when the variable is undefined. *EXT* and *ZIP* are defined only when *-b* is specified as a command-line option (as *tbz* and *j*, respectively). When they have not been defined earlier in the script, then the values *z* and *tgz* are used.

Numeric Conditions

I've now shown examples of both conditions involving string comparisons and file characteristics. Listing 5 introduces numeric conditions; the script is designed for a company president's secretary who wants to check whether someone is logged in.

Listing 5: Adding Numeric Conditions

```
#!/bin/bash

if [ $# -lt 1 ]; then           # No argument given, so prompt
   read -p "Who did you want to check for? " WHO
   if [ -z $WHO ]; then         # No name entered
      exit 0
   fi
else
   WHO="$1"                     # Save the command line argument
fi

LOOK=$(w | grep "^$WHO")
if [ $? -eq 0 ]; then           # Check previous command status
   WHEN=$(echo $LOOK | awk '{print $4}')
   echo "$WHO has been logged in since $WHEN."
else
   echo "$WHO is not currently logged in."
fi
exit 0
```

This script first checks whether any argument was specified on the command line. If not, that is, if the number of argument is less than *1*, then it prompts for the desired user with the *read* command. The user's response is placed into the variable *WHO*. Continuing this first case, if *WHO* is zero length, then the user didn't enter a username but just hit a carriage return, so the script exits. On the other hand, if an argument was specified on the command line, then *WHO* is set to that value. Either way, *WHO* ultimately holds the name of the user to look for.

The second part of the script in Listing 5 uses two command substitution constructs. The first of these constructs searches the output of the *w* command for the desired username, storing the relevant line in *LOOK* if successful. The second defines the variable *WHEN* as the fourth field of that output (the most recent login time), extracting it with *awk* (you don't have to understand everything about awk to use this simple recipe for pulling out a field).

This command runs when *$?* equals 0. *$?* is the status code returned by the most recent command: *grep*. grep returns *0* when it finds a match and *1* otherwise. Finally, the script displays an appropriate message giving the user's status, as in the following example:

```
kyrre has been logged in since 08:47.
```

while and read

The following script illustrates another use of *while* and *read*: processing successive lines of output or a file. The purpose of this script is to send mail messages to a list of (opted-in) users as separate messages:

Table 1: Bash Scripting Quick Summary

Arguments and Variables	
$1 $2 ? $9	Command arguments
${*nn*}	General format for argument *nn*
$@	All command arguments: list of separate items
$*	All command arguments: a single item
$#	Number of command arguments
$0	Script name
$*var*	Value of variable *var*
${*var*}	General format
${*var*:*p*:*n*}	Substring of *n* characters of *var* beginning at *p*
${*var*-*val2*}	Return *val2* if *var* is undefined
${*var*+*val2*}	Return *val2* if *var* is defined
${*var*=*val2*}	Return *val2* if *var* is undefined and set *var*=*val2*
${*var*?*errmsg*}	Display "*var*: *errmsg*" if *var* is undefined
arr=(*items*)	Define *arr* as an array
${*arr*[*n*]}	Element *n* of array *arr*
${#*arr*[@]}	Number of defined elements in *arr*
getopts *opts var*	Process options, returning option letter in *var* (or *?* if invalid, or *:* if required argument is missing); *opts* lists valid option letters optionally followed by a colon to require an argument (an initial colon says to ignore invalid options). Returns option's argument in *OPTARG*.
General Command Constructs	
\`*cmd*\`	Substitute output of *cmd*.
$(*cmd*)	Substitute output of *cmd*.
$?	Exit status of most recent command.
$!	PID of most recently started background command.
eval *string*	Perform substitution operations on *string* and then execute.
. *file*	Include *file* contents within script.
exit *n*	Exit script with status *n* (0 means success).
Arithmetic	
$((*expression*))	Evaluate *expression* as an integer operation.
+ - * /	Addition, subtraction, multiplication, division
++ –	Increment, decrement
%	Modulus
	Exponentiation

Constructing Conditions	
-*x file*	Tests whether *file* has condition indicated by code letter *x*. Some useful codes are: -s greater than *0* length; -r readable; -w writable; -e exists; -d a directory; -f a regular file.
file1-nt *file2*	*file1* is newer than *file2*.
-z *string*	*string*'s length is *0*.
-n *string*	*string*'s length is greater than *0*.
string1 = *string2*	The two strings are identical. Other operations: !=, >, <.
int1-eq *int2*	The two integers are equal. Other operations: -ne, -gt, -lt, -ge, -le.
!	NOT
-a	AND
-o	OR
( )	Used for grouping conditions.
Input and Output	
read *vars*	Read input line and assign successive words to each variable.
read -p *string var*	Prompt for a single value and place value entered into *var*.
printf *fstring vars*	Display the values of *vars* according to the format specified in *fstring*. Format string consists of literal text, possibly including escaped characters (e.g., *t* for tab, *n* for newline) plus format codes. Some of the most useful are: %s for string, %d for signed integers, %f for floating point (%% is a literal percent sign). Follow the percent sign with a hyphen to specify right alignment. You can also precede the code letter with a number to specify a field width. For example, %-5d means a five-digit integer aligned on the right, %6.2f specifies a field width of six with two decimal places for a floating point value.
Functions	

name ()
{
commands
}
	Use local to limit variable scope to the function

```
#!/bin/bash

/bin/cat ➘
/usr/local/sbin/email_list |
while read WHO SUBJ; do
   /usr/bin/mail  ➘
    -s "$SUBJ" $WHO < $WHAT
    echo $WHO
done
```

The script sends the contents of the file names to the *while* command; the condition used here is a *read* command specifying three variables. *read* will process each successive line from *while*'s standard input – the output of the *cat* command – and assigns the first word to *WHO*, the second word to *WHAT*, and all remaining words to *SUBJ* (where words are separated by white space by default). These specify the email address, message file, and subject string for each person. These variables are then used to build the subsequent mail command.

Note that this script uses full pathnames for all external commands. You should adopt the practice of always using full pathnames or including an explicit *PATH* definition at the beginning of the script to avoid security problems from substituted executables. Unfortunately, the script is quite sanguine about trusting the contents of the *email_list* file to include properly formatted email addresses. If such a script is meant for use by someone other than the writer, careful checking of email addresses is necessary. Consider the effect of a username like *jane@ahania.com; /somewhere/run_me* within the address list.

Loops

The next two scripts illustrate other kinds of loops you can use in shell scripts via the *for* command. Listing 6 prepares a report of the total disk space used with a list of directory locations for a set of users. The files containing the list of users and the directories to examine are specified explicitly in the script, but you could also use options for them. The script begins by setting the path and incorporating another file into the script via the so-called dot command include-file mechanism (invoked with a period).

A number of items are notable in this script:

- The *for* command specifies a variable, the keyword *in*, a list of items, and finally the separate command *do*. Each time through the loop (which ends with *done*), the variable is assigned to the next item in the list. *WHO* is assigned to each successive item in the *ckusers* file. The construct *$(< file)* is shorthand for *$(cat file)*.
- The definition of *HOMESUM* uses back quotes to extract the total size of the user's home directory from the output of *du -s* via *awk*. The *eval* command causes *du* to interpret the expanded version of *~$WHO* as a tilde home directory specifier.
- The definition of *TMPLIST* uses command substitution to store the size field (again via awk) from all lines of *ls -lR* output corresponding to items owned by the current user (identified by *egrep*). The *ls* command runs over the directories specified in the *ckdirs* file and uses the *-block-size* option to make its size display unit match that used by *du* (KB). *TMPLIST* is a list of numbers: one per file owned by the current user (*$WHO*).
- The second *for* loop adds the numbers in *TMPLIST* into *TSUM*. The variable is *N*, and the list of items is the value of the *TMPLIST* variable.
- The Bash shell provides built-in integer arithmetic via the construct *$((math-expression))*. The script uses this construct twice.
- The script uses a function named *to_gb* for printing each report line. Bash requires that functions be defined before they are used, so functions are typically stored in external files and invoked with the dot command include-file mechanism. The function is stored in *functions.bash*.

This *to_gb* function is shown in Listing 7. The function begins by defining some local variables. The function will ignore any meaning the names might have in the calling script, and their values will also not be carried back into the calling script. The bulk of the function consists of arithmetic operations, using *$((...))*. Bash provides only integer arithmetic, but I want to display a reasonably accurate size total in gigabytes, so I use a standard trick to extract the integer and remainder parts of the gigabyte value and build the display manually. For example, if I have 2987MB, dividing again by 1024 would yield 2GB. So instead, I divide 2987 by 1000 (*D1* = 2) and then compute 2987 – (2*1000) (*D2* = 987). Then, I print *D1*, a decimal point, and then the first character of *D2*: 2.9.

The *printf* command is used to construct formatted output. It requires a format string followed by variables to be printed. Code letters preceded by percent signs with the format string indicate where the variable contents goes. In this case, *%s* indicates each location and indicates that the variable should be printed as a character string. The *\t* and *\n* within the format string correspond to a tab and a newline character, respectively. You must include the latter explicitly when you want the line to end.

Here is some sample output from this script:

Listing 6: Reporting on Disk Space

```
#!/bin/bash

PATH=/bin:/usr/bin                          # set the path
. /usr/local/sbin/functions.bash             # . f => include file f here

printf "USER\tGB USED\n"                    # print report header line
for WHO in $(</usr/local/sbin/ckusers); do
   HOMESUM=`eval du -s ~$WHO | awk '{print $1}'`
   TMPLIST=$( ls -lR --block-size 1024 $(</usr/local/bin/ckdirs) |
               egrep "^.......... +[0-9]+ $WHO" | awk '{print $5}' )
   TSUM=0
   for N in $TMPLIST; do
      TSUM=$(( $TSUM+$N ))
   done
   TOT=$(( $HOMESUM+$TSUM ))
   to_gb $WHO $TOT
done
```

Listing 7: to_gb

```
to_gb()
{
#  arguments: user usage-in-KB

   local MB D1 D2 USER        # local variables
   USER=$1
   MB=$(( $2/1024 ))          # convert to approx. MB
   D1=$(( $MB/1000 ))         # extract integer GBs
   D2=$(( $MB-($D1*1000) ))   # compute remainder

#  display abcd MB as: a.bcd GB
   printf "%s\t%s\n" $USER $D1.${D2:0:1}
   return
}
```

```
USER    GB USED
aeleen  80.5
kyrre   14.3
munin   0.3
```

Listing 8, which computes factorials, illustrates a kind of *for* loop similar to that found in many programming languages (the syntax is quite similar to C).

The *for* syntax supplies a loop variable, along with its starting value, a loop-continuing condition, and an expression indicating how the variable should be modified after each loop iteration. Here the loop is over the variable *I*, whose starting value is the first script variable. At the end of each iteration, the value of *I* is decreased by 1 (*I++* would similarly increment *I*), and the loop continues as long as *I* is greater than *1*. The body of the loop multiplies *F* (set to *1* initially) by each successive *I*. The script ends by printing the final result:

```
$ ./fact 6
6! = 720
```

Generating Menus

The final script illustrates Bash's built-in menu generation capability via its *select* command (Listing 9). Setup for the *select* command happens in the definitions of *PKGS* and *MENU*. The *select* command requires a list of items as its second argument, and *MENU* will serve that purpose. It is defined via a command substitution construct. Here, I add the literal string *Done* to the end of the list.

The definition of *PKGS* introduces a new feature: arrays. An array is a data structure containing multiple items that can be referenced by an index. The following defines and uses a simple array:

```
$ a=(1 2 3 4 5)
$ echo ${a[2]}
3
```

Listing 8: Factorial Script

```
#!/bin/bash

F=1
for (( I=$1 ; I>1 ; I-- )); do
   F=$(( $F*$I ))
done
echo $1'! = '$F
exit 0
```

An array can be defined by enclosing its elements in parentheses. Specific array elements are specified using the syntax in the second line: The array name is inside the curly braces, and the desired element is specified in square brackets. Note that element numbering begins at *0*. Under normal circumstances, the number of elements in an array is given by *${#a[@]}*. *PKGS* is defined as an array consisting of the second field in each line in the file.

The *select* command uses the contents of *MENU* as its list. It will construct a numbered text menu from the list items and then prompt the user for a selection. The item selected is returned in the variable specified before *in* (here *WHAT*), and the item number is returned in the variable *REPLY*.

The script will use the value of *REPLY* minus *1* to retrieve the corresponding package name from the PKGS array in the variable *PICKED* (I use *$REPLY-1*, because menu numbering begins at *1*, although array element numbering begins at *0*). The *select* command exits when the user picks the *Done* item.

The following is an example run of this script:

```
1) CD/MP3_Player  3) Photo_Album
2) Spider_Solitaire  4) Done
#? 2
Installing package spider ... ↩
   Please be patient!
many more messages ...
#? 4
```

Listing 9: Generating Menus

```
#!/bin/bash

PATH=/bin:/usr/bin
PFILE=/usr/local/sbin/userpkgs                     # entry format: pkgname menu_item

PKGS=( $(cat $PFILE | awk '{print $1}') )      # array of package names
MENU="$(cat $PFILE | awk '{print $2}') Done"   # list of menu items

select WHAT in $MENU; do
   if [ $WHAT = "Done" ]; then exit; fi
   I=$(( $REPLY-1 ))
   PICKED=${PKGS[$I]}
   echo Installing package $PICKED ... Please be patient!
   additional commands to install the package
done
```

Conclusion

See the table titled "Bash Scripting Quick Summary" for a quick reference on Bash scripting terms. I hope you have enjoyed this foray into the world of Bash scripting. You can use these techniques to build your own Bash scripts for automating common tasks. Have a good time with further explorations. ■