Function local variables

Local variables are helpful in a function written for general use. Because the function is called by many scripts that might be written by different programmers, you need to make sure the names of the variables used within the function do not conflict with (i.e., duplicate) the names of the variables in the programs that call the function. Local variables eliminate this problem. When used within a function, the local builtin declares a variable to be local to the function it is defined in.

The next example shows the use of a local variable in a function. It features two variables named **count**. The first is declared and initialized to 10 in the interactive shell. Its value never changes, as echo verifies after **count_down** is run. The other **count** is declared, using local, to be local to the **count_down** function. Its value, which is unknown outside the function, ranges from 4 to 1, as the echo command within the function confirms.

The following example shows the function being entered from the keyboard; it is not a shell script. See the tip "A function is not a shell script" on page 453.

```
$ count_down () {
> local count
> count=$1
> while [ $count -gt 0 ]
> do
> echo "$count..."
> ((count=count-1))
> sleep 1
> done
> echo "Blast Off."
> }

$ count=10
$ count_down 4
4...
3...
2...
1...
Blast Off.
$ echo $count
10
```

The **count=count–1** assignment is enclosed between double parentheses, which cause the shell to perform an arithmetic evaluation (page 492). Within the double parentheses you can reference shell variables without the leading dollar sign (**$**). See page 347 for another example of function local variables.

BUILTIN COMMANDS

Builtin commands, which were introduced in Chapter 5, do not fork a new process when you execute them. This section discusses the type, read, exec, trap, kill, and getopts builtins. Table 10-6 on page 491 lists many bash builtin commands. See Table 9-10 on page 407 for a list of tcsh builtins.

type: DISPLAYS INFORMATION ABOUT A COMMAND

The type builtin (use which under tcsh) provides information about a command:

```
$ type cat echo who if lt
cat is hashed (/bin/cat)
echo is a shell builtin
who is /usr/bin/who
if is a shell keyword
lt is aliased to 'ls -ltrh | tail'
```

The preceding output shows the files that would be executed if you gave **cat** or **who** as a command. Because cat has already been called from the current shell, it is in the hash table (page 325) and type reports that **cat is hashed**. The output also shows that a call to **echo** runs the echo builtin, **if** is a keyword, and **lt** is an alias.

read: ACCEPTS USER INPUT

A common use for user-created variables is storing information that a user enters in response to a prompt. Using read, scripts can accept input from the user and store that input in variables. See page 391 for information about reading user input under tcsh. The read builtin reads one line from standard input and assigns the words on the line to one or more variables:

```
$ cat read1
echo -n "Go ahead: "
read firstline
echo "You entered: $firstline"

$ ./read1
Go ahead: This is a line.
You entered: This is a line.
```

The first line of the **read1** script uses echo to prompt for a line of text. The **–n** option suppresses the following NEWLINE, allowing you to enter a line of text on the same line as the prompt. The second line reads the text into the variable **firstline**. The third line verifies the action of read by displaying the value of **firstline**.

The **–p** (prompt) option causes read to send to standard error the argument that follows it; read does not terminate this prompt with a NEWLINE. This feature allows you to both prompt for and read the input from the user on one line:

```
$ cat read1a
read -p "Go ahead: " firstline
echo "You entered: $firstline"

$ ./read1a
Go ahead: My line.
You entered: My line.
```

The variable in the preceding examples is quoted (along with the text string) because you, as the script writer, cannot anticipate which characters the user might enter in response to the prompt. Consider what would happen if the variable were not quoted and the user entered * in response to the prompt:

```
$ cat read1_no_quote
read -p "Go ahead: " firstline
echo You entered: $firstline

$ ./read1_no_quote
Go ahead: *
You entered: read1 read1_no_quote script.1
$ ls
read1   read1_no_quote    script.1
```

The ls command lists the same words as the script, demonstrating that the shell expands the asterisk into a list of files in the working directory. When the variable **$firstline** is surrounded by double quotation marks, the shell does not expand the asterisk. Thus the **read1** script behaves correctly:

```
$ ./read1
Go ahead: *
You entered: *
```

REPLY When you do not specify a variable to receive read's input, bash puts the input into the variable named **REPLY**. The following **read1b** script performs the same task as **read1**:

```
$ cat read1b
read -p "Go ahead: "
echo "You entered: $REPLY"
```

The **read2** script prompts for a command line, reads the user's response, and assigns it to the variable **cmd**. The script then attempts to execute the command line that results from the expansion of the **cmd** variable:

```
$ cat read2
read -p "Enter a command: " cmd
$cmd
echo "Thanks"
```

In the following example, **read2** reads a command line that calls the echo builtin. The shell executes the command and then displays **Thanks**. Next **read2** reads a command line that executes the who utility:

```
$ ./read2
Enter a command: echo Please display this message.
Please display this message.
Thanks
$ ./read2
Enter a command: who
max       pts/4         2013-06-17 07:50  (:0.0)
sam       pts/12        2013-06-17 11:54  (guava)
Thanks
```

If **cmd** does not expand into a valid command line, the shell issues an error message:

```
$ ./read2
Enter a command: xxx
./read2: line 2: xxx: command not found
Thanks
```

The **read3** script reads values into three variables. The read builtin assigns one word (a sequence of nonblank characters) to each variable:

```
$ cat read3
read -p "Enter something: " word1 word2 word3
echo "Word 1 is: $word1"
echo "Word 2 is: $word2"
echo "Word 3 is: $word3"
$ ./read3
Enter something: this is something
Word 1 is: this
Word 2 is: is
Word 3 is: something
```

When you enter more words than read has variables, read assigns one word to each variable, assigning all leftover words to the last variable. Both **read1** and **read2** assigned the first word and all leftover words to the one variable the scripts each had to work with. In the following example, read assigns five words to three variables: It assigns the first word to the first variable, the second word to the second variable, and the third through fifth words to the third variable.

```
$ ./read3
Enter something: this is something else, really.
Word 1 is:  this
Word 2 is:  is
Word 3 is:  something else, really.
```

Table 10-4 lists some of the options supported by the read builtin.

Table 10-4 read options

Option	Function
–a ***aname***	(array) Assigns each word of input to an element of array ***aname***.
–d ***delim***	(delimiter) Uses ***delim*** to terminate the input instead of NEWLINE.
–e	(Readline) If input is coming from a keyboard, uses the Readline Library (page 335) to get input.
–n ***num***	(number of characters) Reads ***num*** characters and returns. As soon as the user types ***num*** characters, read returns; there is no need to press RETURN.
–p ***prompt***	(prompt) Displays ***prompt*** on standard error without a terminating NEWLINE before reading input. Displays ***prompt*** only when input comes from the keyboard.
–s	(silent) Does not echo characters.

–u*n*	(file descriptor) Uses the integer ***n*** as the file descriptor that read takes its input from. Thus
	`read –u4 arg1 arg2`
	is equivalent to
	`read arg1 arg2 <&4`
	See "File Descriptors" (page 452) for a discussion of redirection and file descriptors.

The read builtin returns an exit status of 0 if it successfully reads any data. It has a nonzero exit status when it reaches the EOF (end of file).

The following example runs a **while** loop from the command line. It takes its input from the **names** file and terminates after reading the last line from **names**.

```
$ cat names
Alice Jones
Robert Smith
Alice Paulson
John Q. Public

$ while read first rest
> do
> echo $rest, $first
> done < names
Jones, Alice
Smith, Robert
Paulson, Alice
Q. Public, John
$
```

The placement of the redirection symbol (**<**) for the **while** structure is critical. It is important that you place the redirection symbol at the **done** statement and not at the call to read.

optional Each time you redirect input, the shell opens the input file and repositions the read pointer at the start of the file:

```
$ read line1 < names; echo $line1; read line2 < names; echo $line2
Alice Jones
Alice Jones
```

Here each read opens **names** and starts at the beginning of the **names** file. In the following example, **names** is opened once, as standard input of the subshell created by the parentheses. Each read then reads successive lines of standard input:

```
$ (read line1; echo $line1; read line2; echo $line2) < names
Alice Jones
Robert Smith
```

Another way to get the same effect is to open the input file with exec and hold it open (refer to "File Descriptors" on page 452):

```
$ exec 3< names
$ read -u3 line1; echo $line1; read -u3 line2; echo $line2
Alice Jones
Robert Smith
$ exec 3<&-
```

exec: EXECUTES A COMMAND OR REDIRECTS FILE DESCRIPTORS

The exec builtin (not in tcsh) has two primary purposes: to run a command without creating a new process and to redirect a file descriptor—including standard input,

output, or error—of a shell script from within the script (page 452). When the shell executes a command that is not built into the shell, it typically creates a new process. The new process inherits environment (exported) variables from its parent but does not inherit variables that are not exported by the parent (page 468). In contrast, exec executes a command in place of (overlays) the current process.

exec: EXECUTES A COMMAND

The exec builtin used for running a command has the following syntax:

exec command arguments

exec versus . (dot)

Insofar as exec runs a command in the environment of the original process, it is similar to the . (dot) command (page 280). However, unlike the . command, which can run only shell scripts, exec can run both scripts and compiled programs. Also, whereas the . command returns control to the original script when it finishes running, exec does not. Finally the . command gives the new program access to local variables, whereas exec does not.

exec does not return control

Because the shell does not create a new process when you use exec, the command runs more quickly. However, because exec does not return control to the original program, it can be used only as the last command in a script. The following script shows that control is not returned to the script:

```
$ cat exec_demo
who
exec date
echo "This line is never displayed."

$ ./exec_demo
zach     pts/7    May 20  7:05 (guava)
hls      pts/1    May 20  6:59 (:0.0)
Fri May 24 11:42:56 PDT 2013
```

The next example, a modified version of the **out** script (page 425), uses exec to execute the final command the script runs. Because **out** runs either cat or less and then terminates, the new version, named **out2**, uses exec with both cat and less:

```
$ cat out2
if [ $# -eq 0 ]
    then
        echo "Usage: out2 [-v] filenames" 1>&2
        exit 1
fi
if [ "$1" = "-v" ]
    then
        shift
        exec less "$@"
    else
        exec cat -- "$@"
fi
```

exec: REDIRECTS INPUT AND OUTPUT

The second major use of exec is to redirect a file descriptor—including standard input, output, or error—from within a script. The next command causes all subsequent input to a script that would have come from standard input to come from the file named **infile**:

```
exec < infile
```

Similarly the following command redirects standard output and standard error to **outfile** and **errfile**, respectively:

```
exec > outfile 2> errfile
```

When you use exec in this manner, the current process is not replaced with a new process and exec can be followed by other commands in the script.

/dev/tty When you redirect the output from a script to a file, you must make sure the user sees any prompts the script displays. The **/dev/tty** device is a pseudonym for the screen the user is working on; you can use this device to refer to the user's screen without knowing which device it is. (The tty utility displays the name of the device you are using.) By redirecting the output from a script to **/dev/tty**, you ensure that prompts and messages go to the user's terminal, regardless of which terminal the user is logged in on. Messages sent to **/dev/tty** are also not diverted if standard output and standard error from the script are redirected.

The **to_screen1** script sends output to three places: standard output, standard error, and the user's screen. When run with standard output and standard error redirected, **to_screen1** still displays the message sent to **/dev/tty** on the user's screen. The **out** and **err** files hold the output sent to standard output and standard error, respectively.

```
$ cat to_screen1
echo "message to standard output"
echo "message to standard error" 1>&2
echo "message to screen" > /dev/tty

$ ./to_screen1 > out 2> err
message to screen
$ cat out
message to standard output
$ cat err
message to standard error
```

The following command redirects standard output from a script to the user's screen:

```
exec > /dev/tty
```

Putting this command at the beginning of the previous script changes where the output goes. In **to_screen2**, exec redirects standard output to the user's screen so the **> /dev/tty** is superfluous. Following the exec command, all output sent to standard output goes to **/dev/tty** (the screen). Output to standard error is not affected.

```
$ cat to_screen2
exec > /dev/tty
echo "message to standard output"
echo "message to standard error" 1>&2
echo "message to screen" > /dev/tty

$ ./to_screen2 > out 2> err
message to standard output
message to screen
```

One disadvantage of using exec to redirect the output to **/dev/tty** is that all subsequent output is redirected unless you use exec again in the script.

You can also redirect the input to read (standard input) so that it comes from **/dev/tty** (the keyboard):

```
read name < /dev/tty
```

or

```
exec < /dev/tty
```

trap: CATCHES A SIGNAL

A *signal* is a report to a process about a condition. Linux uses signals to report interrupts generated by the user (for example, pressing the interrupt key) as well as bad system calls, broken pipelines, illegal instructions, and other conditions. The trap builtin (tcsh uses onintr) catches (traps) one or more signals, allowing you to direct the actions a script takes when it receives a specified signal.

This discussion covers six signals that are significant when you work with shell scripts. Table 10-5 lists these signals, the signal numbers that systems often ascribe to them, and the conditions that usually generate each signal. Give the command **kill –l** (lowercase "l"), **trap –l** (lowercase "l"), or **man 7 signal** to display a list of all signal names.

Table 10-5 Signals

Type	Name	Number	Generating condition
Not a real signal	EXIT	0	Exit because of exit command or reaching the end of the program (not an actual signal but useful in trap)
Hang up	SIGHUP or HUP	1	Disconnect the line
Terminal interrupt	SIGINT or INT	2	Press the interrupt key (usually CONTROL-C)

Table 10-5 Signals (continued)

Type	Name	Number	Generating condition
Quit	SIGQUIT or QUIT	3	Press the quit key (usually CONTROL-SHIFT-| or CONTROL-SHIFT-\)
Kill	SIGKILL or KILL	9	The kill builtin with the **–9** option (cannot be trapped; use only as a last resort)
Software termination	SIGTERM or TERM	15	Default of the kill command
Stop	SIGTSTP or TSTP	20	Press the suspend key (usually CONTROL-Z)
Debug	DEBUG		Execute ***commands*** specified in the trap statement after each command (not an actual signal but useful in trap)
Error	ERR		Execute ***commands*** specified in the trap statement after each command that returns a nonzero exit status (not an actual signal but useful in trap)

When it traps a signal, a script takes whatever action you specify: It can remove files or finish other processing as needed, display a message, terminate execution immediately, or ignore the signal. If you do not use trap in a script, any of the six actual signals listed in Table 10-5 (not EXIT, DEBUG, or ERR) will terminate the script. Because a process cannot trap a KILL signal, you can use **kill –KILL** (or **kill –9**) as a last resort to terminate a script or other process. (See page 486 for more information on kill.)

The trap command has the following syntax:

*trap ['**commands**'] [**signal**]*

The optional ***commands*** specifies the commands the shell executes when it catches one of the signals specified by ***signal***. The ***signal*** can be a signal name or number—for example, INT or 2. If ***commands*** is not present, trap resets the trap to its initial condition, which is usually to exit from the script.

Quotation marks

The trap builtin does not require single quotation marks around ***commands*** as shown in the preceding syntax but it is a good practice to use them. The single quotation marks cause shell variables within the ***commands*** to be expanded when the signal occurs, rather than when the shell evaluates the arguments to trap. Even if you do not use any shell variables in the ***commands***, you need to enclose any

command that takes arguments within either single or double quotation marks. Quoting ***commands*** causes the shell to pass to trap the entire command as a single argument.

After executing the ***commands***, the shell resumes executing the script where it left off. If you want trap to prevent a script from exiting when it receives a signal but not to run any commands explicitly, you can specify a null (empty) ***commands*** string, as shown in the **locktty** script (page 440). The following command traps signal number 15, after which the script continues:

```
trap '' 15
```

The following script demonstrates how the trap builtin can catch the terminal interrupt signal (2). You can use SIGINT, INT, or 2 to specify this signal. The script returns an exit status of 1:

```
$ cat inter
#!/bin/bash
trap 'echo PROGRAM INTERRUPTED; exit 1' INT
while true
do
    echo "Program running."
    sleep 1
done
$ ./inter
Program running.
Program running.
Program running.
CONTROL-C
PROGRAM INTERRUPTED
$
```

: (null) builtin

The second line of **inter** sets up a trap for the terminal interrupt signal using INT. When trap catches the signal, the shell executes the two commands between the single quotation marks in the trap command. The echo builtin displays the message **PROGRAM INTERRUPTED**, exit terminates the shell running the script, and the parent shell displays a prompt. If exit were not there, the shell would return control to the **while** loop after displaying the message. The **while** loop repeats continuously until the script receives a signal because the true utility always returns a *true* exit status. In place of true you can use the **:** (null) builtin, which is written as a colon and always returns a 0 (*true*) status.

The trap builtin frequently removes temporary files when a script is terminated prematurely, thereby ensuring the files are not left to clutter the filesystem. The following shell script, named **addbanner**, uses two traps to remove a temporary file when the script terminates normally or because of a hangup, software interrupt, quit, or software termination signal:

```
$ cat addbanner
#!/bin/bash
script=$(basename $0)

if [ ! -r "$HOME/banner" ]
    then
        echo "$script: need readable $HOME/banner file" 1>&2
        exit 1
fi

trap 'exit 1' 1 2 3 15
trap 'rm /tmp/$$.$script 2> /dev/null' EXIT

for file
do
    if [ -r "$file" -a -w "$file" ]
        then
            cat $HOME/banner $file > /tmp/$$.$script
            cp /tmp/$$.$script $file
            echo "$script: banner added to $file" 1>&2
        else
            echo "$script: need read and write permission for $file" 1>&2
        fi
done
```

When called with one or more filename arguments, **addbanner** loops through the files, adding a header to the top of each. This script is useful when you use a standard format at the top of your documents, such as a standard layout for memos, or when you want to add a standard header to shell scripts. The header is kept in a file named **~/banner**. Because **addbanner** uses the **HOME** variable, which contains the pathname of the user's home directory, the script can be used by several users without modification. If Max had written the script with **/home/max** in place of **$HOME** and then given the script to Zach, either Zach would have had to change it or **addbanner** would have used Max's **banner** file when Zach ran it (assuming Zach had read permission for the file).

The first trap in **addbanner** causes it to exit with a status of 1 when it receives a hangup, software interrupt (terminal interrupt or quit signal), or software termination signal. The second trap uses **EXIT** in place of **signal-number**, which causes trap to execute its command argument *whenever* the script exits because it receives an exit command or reaches its end. Together these traps remove a temporary file whether the script terminates normally or prematurely. Standard error of the second trap is sent to **/dev/null** whenever trap attempts to remove a nonexistent temporary file. In those cases rm sends an error message to standard error; because standard error is redirected, the user does not see the message.

See page 440 for another example that uses trap.

kill: ABORTS A PROCESS

The kill builtin sends a signal to a process or job. The kill command has the following syntax:

kill [–signal] PID

where ***signal*** is the signal name or number (for example, INT or 2) and ***PID*** is the process identification number of the process that is to receive the signal. You can specify a job number (page 146) as ***%n*** in place of ***PID***. If you omit ***signal***, kill sends a TERM (software termination, number 15) signal. For more information on signal names and numbers, see Table 10-5 on page 483.

The following command sends the TERM signal to job number 1, regardless of whether it is running or stopped in the background:

```
$ kill -TERM %1
```

Because TERM is the default signal for kill, you can also give this command as **kill %1**. Give the command **kill –l** (lowercase "l") to display a list of signal names.

A program that is interrupted can leave matters in an unpredictable state: Temporary files might be left behind (when they are normally removed), and permissions might be changed. A well-written application traps signals and cleans up before exiting. Most carefully written applications trap the INT, QUIT, and TERM signals.

To terminate a program, first try INT (press CONTROL-C, if the job running is in the foreground). Because an application can be written to ignore this signal, you might need to use the KILL signal, which cannot be trapped or ignored; it is a "sure kill." Refer to page 846 for more information on kill. See also the related utility killall (page 848).

eval: SCANS, EVALUATES, AND EXECUTES A COMMAND LINE

The eval builtin scans the command that follows it on the command line. In doing so, eval processes the command line in the same way bash does when it executes a command line (e.g., it expands variables, replacing the name of a variable with its value). For more information refer to "Processing the Command Line" on page 354. After scanning (and expanding) the command line, it passes the resulting command line to bash to execute.

The following example first assigns the value **frog** to the variable **name**. Next eval scans the command **$name=88** and expands the variable **$name** to **frog**, yielding the command **frog=88**, which it passes to bash to execute. The last command displays the value of **frog**.

```
$ name=frog
$ eval $name=88
$ echo $frog
88
```

Brace expansion with a sequence expression

The next example uses eval to cause brace expansion with a sequence expression (page 357) to accept variables, which it does not normally do. The following command demonstrates brace expansion with a sequence expression:

```
$ echo {2..5}
2 3 4 5
```

One of the first things bash does when it processes a command line is to perform brace expansion; later it expands variables (page 354). When you provide an invalid argument in brace expansion, bash does not perform brace expansion; instead, it passes the string to the program being called. In the next example, bash cannot expand {**$m..$n**} during the brace expansion phase because it contains variables, so it continues processing the command line. When it gets to the variable expansion phase, it expands **$m** and **$n** and then passes the string {**2..5**} to echo.

```
$ m=2 n=5
$ echo {$m..$n}
{2..5}
```

When eval scans the same command line, it expands the variables as explained previously and yields the command **echo** {**2..5**}. It then passes that command to bash, which can now perform brace expansion:

```
$ eval echo {$m..$n}
2 3 4 5
```

getopts: PARSES OPTIONS

The getopts builtin (not in tcsh) parses command-line arguments, making it easier to write programs that follow the Linux argument conventions. The syntax for getopts is

getopts ***optstring varname*** *[**arg** ...]*

where ***optstring*** is a list of the valid option letters, ***varname*** is the variable that receives the options one at a time, and ***arg*** is the optional list of parameters to be processed. If ***arg*** is not present, getopts processes the command-line arguments. If ***optstring*** starts with a colon (**:**), the script must take care of generating error messages; otherwise, getopts generates error messages.

The getopts builtin uses the **OPTIND** (option index) and **OPTARG** (option argument) variables to track and store option-related values. When a shell script starts, the value of **OPTIND** is 1. Each time getopts is called and locates an argument, it increments **OPTIND** to the index of the next option to be processed. If the option takes an argument, bash assigns the value of the argument to **OPTARG**.

To indicate that an option takes an argument, follow the corresponding letter in ***optstring*** with a colon (**:**). For example, the ***optstring*** **dxo:lt:r** instructs getopts to search for the **–d**, **–x**, **–o**, **–l**, **–t**, and **–r** options and tells it the **–o** and **–t** options take arguments.

Using getopts as the ***test-command*** in a **while** control structure allows you to loop over the options one at a time. The getopts builtin checks the option list for options that are in ***optstring***. Each time through the loop, getopts stores the option letter it finds in ***varname***.

As an example, assume you want to write a program that can take three options:

1. A **–b** option indicates that the program should ignore whitespace at the start of input lines.
2. A **–t** option followed by the name of a directory indicates that the program should store temporary files in that directory. Otherwise, it should use **/tmp**.
3. A **–u** option indicates that the program should translate all output to uppercase.

In addition, the program should ignore all other options and end option processing when it encounters two hyphens (**––**).

The problem is to write the portion of the program that determines which options the user has supplied. The following solution does not use getopts:

```
SKIPBLANKS=
TMPDIR=/tmp
CASE=lower
while [[ "$1" = -* ]] # [[ = ]] does pattern match
do
    case $1 in
        -b)     SKIPBLANKS=TRUE ;;
        -t)     if [ -d "$2" ]
                    then
                    TMPDIR=$2
                    shift
                else
                    echo "$0: -t takes a directory argument." >&2
                    exit 1
                fi ;;
        -u)     CASE=upper ;;
        --)     break   ;;      # Stop processing options
        *)      echo "$0: Invalid option $1 ignored." >&2 ;;
        esac
    shift
done
```

This program fragment uses a loop to check and shift arguments while the argument is not **––**. As long as the argument is not two hyphens, the program continues to loop through a **case** statement that checks for possible options. The **–– case** label breaks out of the **while** loop. The *** case** label recognizes any option; it appears as the last **case** label to catch any unknown options, displays an error message, and allows processing to continue. On each pass through the loop, the program uses shift so it accesses the next argument on the next pass through the loop. If an option takes an argument, the program uses an extra shift to get past that argument.

The following program fragment processes the same options using getopts:

```
SKIPBLANKS=
TMPDIR=/tmp
CASE=lower

while getopts :bt:u arg
do
    case $arg in
        b)      SKIPBLANKS=TRUE ;;
        t)      if [ -d "$OPTARG" ]
                    then
                    TMPDIR=$OPTARG
                else
                    echo "$0: $OPTARG is not a directory." >&2
                    exit 1
                fi ;;
        u)      CASE=upper ;;
        :)      echo "$0: Must supply an argument to -$OPTARG." >&2
                exit 1 ;;
        \?)     echo "Invalid option -$OPTARG ignored." >&2 ;;
        esac
done
```

In this version of the code, the **while** structure evaluates the getopts builtin each time control transfers to the top of the loop. The getopts builtin uses the **OPTIND** variable to keep track of the index of the argument it is to process the next time it is called. There is no need to call shift in this example.

In the getopts version of the script, the **case** patterns do not start with a hyphen because the value of **arg** is just the option letter (getopts strips off the hyphen). Also, getopts recognizes `--` as the end of the options, so you do not have to specify it explicitly, as in the **case** statement in the first example.

Because you tell getopts which options are valid and which require arguments, it can detect errors in the command line and handle them in two ways. This example uses a leading colon in ***optstring*** to specify that you check for and handle errors in your code; when getopts finds an invalid option, it sets ***varname*** to **?** and **OPTARG** to the option letter. When it finds an option that is missing an argument, getopts sets ***varname*** to **:** and **OPTARG** to the option lacking an argument.

The **\? case** pattern specifies the action to take when getopts detects an invalid option. The **: case** pattern specifies the action to take when getopts detects a missing option argument. In both cases getopts does not write any error message but rather leaves that task to you.

If you omit the leading colon from ***optstring***, both an invalid option and a missing option argument cause ***varname*** to be assigned the string **?**. **OPTARG** is not set and getopts writes its own diagnostic message to standard error. Generally this method is less desirable because you have less control over what the user sees when an error occurs.

Using getopts will not necessarily make your programs shorter. Its principal advantages are that it provides a uniform programming interface and that it enforces standard option handling.