16.13 set AND shift: MANIPULATING THE POSITIONAL PARAMETERS

Some UNIX commands like **date** produce single-line output. We also pass command output through filters like **grep** and **head** to produce a single line. Recall that we had to use an external command (**cut**) to extract a field from the **date** output *(16.8)*:

```
case `date | cut -d" " -f1` in
```

This is overkill; the shell has an internal command to do this job—the **set** statement. It assigns the positional parameters $1, $2, and so on, to its arguments. This feature is especially useful for picking up individual fields from the output of a program. But before we do that, let's use **set** to convert its arguments to positional parameters:

```
$ set 9876 2345 6213
$ _
```

This assigns the value 9876 to the positional parameter $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $*. You can verify this by echoing each parameter in turn:

```
$ echo "\$1 is $1, \$2 is $2, \$3 is $3"
$1 is 9876, $2 is 2345, $3 is 6213
$ echo "The $# arguments are $*"
The 3 arguments are 9876 2345 6213
```

We'll now use command substitution to extract individual fields from the **date** output without using **cut**:

```
$ set `date`
$ echo $*
Wed Jan 8 09:40:35 IST 2003
$ echo "The date today is $2 $3, $6"
The date today is Jan 8, 2003
```

The day of the week is available in $1. Using the **set** feature, the **case** construct simplifies to **case $1 in**.

Tip **set** parses its arguments on the delimiters specified in the environment variable IFS, which, by default, is whitespace. You can change the value of this variable to make **set** work on a different delimiter. This means that you can easily extract any field in a line from /etc/passwd without using **cut**! One of our sample validation scripts *(16.7.1)* makes use of this feature.

16.13.1 shift: Shifting Arguments Left

Many scripts use the first argument to indicate a separate entity—say a filename. The other arguments could then represent a series of strings—probably different patterns to be selected from a file. For this to be possible, for should start its iteration from the second parameter onwards. This is possible with the **shift** statement.

shift transfers the contents of a positional parameter to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes $2, and so on. Try this on the positional parameters filled up with the **date** command:

```
$ echo "$@"                                    Here, "$@" and $* are interchangeable
Wed Jan 8 09:48:44 IST 2003
$ echo $1 $2 $3
Wed Jan 8
$ shift
$ echo $1 $2 $3
Jan 8 09:48:44
$ shift 2                                     Shifts 2 places
$ echo $1 $2 $3
09:48:44 IST 2003
```

Note that the contents of the leftmost parameter, $1, are lost every time **shift** is invoked. In this way, you can access $10 by first shifting it and converting it to $9. So if a script uses twelve arguments, you can shift thrice and then use the ninth parameter. We require the set–shift duo for the next script, **emp7.sh** (Fig. 16.13), which is run with a filename and a set of patterns as arguments.

```
#!/bin/sh
# emp7.sh: Script using shift -- Saves first argument; for works with rest
#
case $# in
    0|1) echo "Usage: $0 file pattern(s)" ; exit 2 ;;
      *) flname=$1                  # Store $1 as a variable before it gets lost
         shift
         for pattern in "$@" ; do       # Starts iteration with $2
             grep "$pattern" $flname || echo "Pattern $pattern not found"
         done ;;
esac
```

Fig. 16.13 emp7.sh

Since the script requires at least two arguments, you should verify their presence before you act on the patterns. We stored $1 in the variable flname because the next **shift** operation would throw it away. Now you can use the script with a variable number of arguments (not less than 2):

```
$ emp7.sh emp.lst
Insufficient number of arguments
$ emp7.sh emp.lst saxena 1006 9876
2345|j.b. saxena       |g.m.      |marketing |12/03/45|8000
1006|chanchal singhvi  |director  |sales     |03/09/38|6700
Pattern 9876 not found
```

Here flname stores emp.lst, and the **for** loop iterates with the three strings, saxena, 1006, 9876.

Tip Every time you use **shift**, the leftmost variable gets lost; so it should be saved in a variable before using **shift**. If you have to start iteration from the fourth parameter, save the first three parameters and then use **shift 3**.

16.13.2 set --: Helps Command Substitution

You'll often need to use **set** with command substitution. There's a small problem though, especially when the output of the command begins with a -:

```
$ set `ls -l unit01`
-rw-r--r--: bad option(s)
```

Since the permissions string begins with a - (for regular files), **set** interprets it as an option and finds it to be a "bad" one. **set** creates another problem when its arguments evaluate to a null string. Consider this command:

```
set `grep PPP /etc/passwd`
```

If the string PPP can't be located in the file, **set** will operate with no arguments and puzzle the user by displaying all variables on the terminal (its default output)! The solution to both these problems lies in the use of -- (two hyphens) immediately after **set**:

```
set -- `ls -l unit01`
set -- `grep PPP /etc/passwd`
```

The first - now taken care of
Null output is no problem

set now understands that the arguments following -- are not to be treated as options. The two hyphens also direct **set** to suppress its default behavior if the arguments evaluate to a null string.

16.14 THE *HERE DOCUMENT* (<<)

There are occasions when the data your program reads is fixed and fairly limited. The shell uses the << symbols to read data from the same file containing the script. This is referred to as a *here document*, signifying that the data is here rather than in a separate file. Any command using standard input can also take input from a here document.

This feature is useful when used with commands that don't accept a filename as argument (like the **mailx** command, for instance). If the message is short (which any mail message is normally expected to be), you can have both the command and message in the same script:

```
mailx sharma << MARK
Your program for printing the invoices has been executed
on `date`. Check the print queue
The updated file is known as $flname
MARK
```

Command substitution permitted
Variable evaluation too
No spaces permitted here

The here document symbol (<<) is followed by three lines of data and a delimiter (the string MARK). The shell treats every line following the command and delimited by MARK as input to the command. sharma at the other end will see the three lines of message text with the date inserted by command substitution and the evaluated filename. The word MARK itself doesn't show up. When this sequence is placed inside a script, execution is faster because **mailx** doesn't have to read an external file; it's here.

Note The contents of a here document are interpreted and processed by the shell before it goes as input to a command. This means you can use command substitution and variables in its input. You can't do that with normal standard input.

16.14.1 Using the Here Document with Interactive Programs

Many commands require input from the user. Often, it's the same input that is keyed in response to a series of questions posed by the command. For instance, you may have to enter a y two or three times when the command pauses, but the questions may not come in quick succession. Rather than wait for the prompt, we can instruct the script to take input from a here document.

We'll now attempt something that has far-reaching consequences. Recall that we used an interactive script **emp1.sh** *(16.2)* by keying in two parameters. We can make that script work noninteractively by supplying the inputs through a here document:

```
$ emp1.sh << END
> director
> emp.lst
> END
Enter the pattern to be searched: Enter the file to be used: Searching for direc
tor from file emp.lst
9876|jai sharma        |director |production|12/03/50|7000
2365|barun sengupta    |director |personnel |11/05/47|7800
Selected records shown above
```

Even though the prompts are displayed in a single line, the important thing is that the script worked. We have been able to run an interactive script noninteractively!

Tip If you write a script that uses the **read** statement, and which often assumes a predefined set of replies, you can make the script behave noninteractively by supplying its input from a here document.

16.15 trap: INTERRUPTING A PROGRAM

By default, shell scripts terminate whenever the interrupt key is pressed. It's not always a good idea to terminate shell scripts in this way because that can leave a lot of temporary files on disk. The **trap** statement lets you do the things you want in case the script receives a signal. The statement is normally placed at the beginning of a shell script and uses two lists:

```
trap 'command_list' signal_list
```

When a script is sent any of the signals in *signal_list*, **trap** executes the commands in *command_list*. The signal list can contain the integer values or names (without the SIG prefix) of one or more signals—the ones you use with the **kill** command. So instead of using 2 15 to represent the signal list, you can also use INT TERM (the recommended approach).

If you habitually create temporary files named after the PID number of the shell, you should use the services of **trap** to remove them whenever an interrupt occurs:

```
trap 'rm $$* ; echo "Program interrupted" ; exit' HUP INT TERM
```

trap is a signal handler. Here, it first removes all files expanded from $$*, echoes a message and finally terminates the script when the signals SIGHUP (1), SIGINT (2) or SIGTERM (15) are sent to the shell process running the script. When the interrupt key is pressed, it sends the signal number 2. It's a good idea to include this number in your scripts.

You may also like to ignore the signal and continue processing. In that case, you should make the script ignore such signals by using a null command list. A script containing the following **trap** statement will not be affected by three signals; this time we'll use the signal numbers:

Script can't be killed by normal means

```
trap '' 1 2 15
```

The Korn and Bourne shells don't execute a file on logging out, but using **trap**, you can make them do that. You'll have to use the signal name EXIT (or 0) as a component of the signal list. These shells also use the statement **trap -** to reset the signals to their default values. You can also use multiple **trap** commands in a script; each one overrides the previous one.

Note It's not mandatory to have a **trap** statement in your shell scripts. However, if you have one, don't forget to include the **exit** statement at the end of the command list unless you want the script to ignore the specific signals.

16.16 DEBUGGING SHELL SCRIPTS WITH set -x

Apart from assigning values to positional parameters, **set** serves as a useful debugging tool with its **-x** option. When used inside a script (or even at the $ prompt), it echoes each statement on the terminal, preceded by a + as it is executed. Modify any previous script to turn on the **set** option by placing the following statement at the beginning of the script:

```
set -x
```

set +x turns off **set -x**, and you can place the latter statement at the end of the script. This is what you'll see when you invoke the script, **emp7.sh**, in the following manner:

```
$ emp7.sh emp.lst 2233 1265 0110
+ flname=emp.lst
+ shift
+ grep 2233 emp.lst
2233|a.k. shukla      |g.m.      |sales      |12/12/52|6000
+ grep 1265 emp.lst
1265|s.n. dasgupta    |manager   |sales      |12/09/63|5600
+ grep 0110 emp.lst
0110|v.k. agrawal     |g.m.      |marketing  |31/12/40|9000
```

This is an ideal tool to use if you have trouble finding out why scripts don't work in the manner expected. Note how the shell prints each statement as it is being executed, affixing a + to each. It even shows you what the **grep** command line looks like at every iteration!

16.17 SAMPLE VALIDATION AND DATA ENTRY SCRIPTS

It's time to consolidate our knowledge by devising two interactive shell scripts that accept input from the user. One script looks up a code list while the other adds an entry to a text database.