24

Advanced Shell Programming

You have already discovered the basic features of the shell—both as an interpreter and as a scripting language. But the shell is more than just an interpreter or a language. It is also a process, an environment which makes itself available to the programs that it executes. It is necessary that you understand the environmental changes that take place when the shell executes a program, especially a shell script. You should also know how to change these environmental parameters.

Advanced knowledge of shell programming is needed by the system administrator, who has to constantly devise scripts that monitor and correct system functioning. A detailed knowledge of the shell's subtle features is also necessary if you aspire to be an expert script writer. The following discussions mostly assume the Bourne shell, but the special features of the Korn and Bash shells are also examined here.

WHAT YOU WILL LEARN

- How the shell uses a second shell to execute a shell script.
- When and how commands are executed in the current shell and a sub-shell.
- Make variables visible in sub-shells with **export**.
- Use **let** and (( )) for computation in the Korn shell and Bash.
- Use arrays supported by the Korn shell and Bash.
- Handle strings with the built-in facilities of the Korn shell and Bash.
- Evaluate variables in different ways depending on whether they are set or not.
- Merge the standard output and standard error streams with the symbols 1>&2 and 2>&1.
- Understand how *shell functions* are superior to aliases.
- Use **eval** to evaluate a command line twice and produce generalized prompts and variables.
- Overlay the current program with another and handle multiple streams with **exec**.

TOPICS OF SPECIAL INTEREST

- The situations that demand execution of a command in the current shell. Use of the dot command for this purpose.
- Devise useful shell functions required for everyday use.
- Access files with file descriptors, using **exec**, in the same way system calls and library functions access them.

24.1 SHELLS AND SUB-SHELLS

When the shell executes a shell script, it first spawns a sub-shell, which in turn executes the commands in the script. When script execution is complete, the child shell withers away and returns control to the parent shell. You can also *explicitly* invoke a sub-shell to execute a shell script. The command representing the shell itself (**sh**, **ksh** or **bash**) can be used to read the statements in **join.sh**:

```
sh join.sh                    Shell also accepts a script name as argument
sh < join.sh                  It's standard input can also be redirected
```

Thus a shell script run with **sh**, **ksh** or **bash** need not have executable permission. This technique, however, is applicable for executing only shell scripts and not executables. You certainly can't use **sh < a.out**.

Even though the shell accepts a script name as argument, we generally don't run shell scripts in the way shown above. We simply key in the script name from the shell prompt and run it as an executable. In this case, the current shell uses a sub-shell of the same type to execute it. However, if the script contains the interpreter line in this form:

```
#!/usr/bin/ksh
```

then, even though the login shell may be Bourne, it will use the Korn shell to execute the script. Specification of the interpreter line also helps us identify the shell the script is meant to use. We have specified the interpreter line in every script used in Chapter 16; we'll continue this practice in this chapter also.

24.2 ( ) AND { }: SUB-SHELL OR CURRENT SHELL?

The shell uses two types of operators to group commands. You must understand clearly the consequences of using one in preference to the other:

- The ( ) Statements enclosed within parentheses are executed in a sub-shell.
- The { } Statements enclosed within curly braces are executed in the current shell only.

You have used the first type *(9.5.2)* to collectively redirect the standard output of two commands with a single redirection symbol in a manner similar to this:

```
( a.sh ; b.sh ; c.sh ) > d.sh
```

Sub-shell considerations are not important here, so we can use either form, but some applications require us to run a set of commands *without* spawning a child shell. To consider an example, let's use both grouping operators with the **cd** and **pwd** commands. Check your current directory and then change it with **cd**:

```
$ pwd
/home/kumar
$ ( cd progs ; pwd )
/home/kumar/progs
$ pwd                         Back to original directory
/home/kumar
```

Working from a sub-shell, **cd** changed the working directory (one of the environmental parameters) to /home/kumar/progs. The parent (login shell) can't adopt this change, so the original directory is back in place. The same command group—this time using the {} operators—tells a different story:

```
$ pwd
/home/kumar
$ { cd progs ; pwd ; }
/home/kumar/progs
$ pwd
/home/kumar/progs                    Directory change is now permanent
```

The two commands have now been executed without spawning a shell; no separate environment was created, and the change of directory became permanent even after the execution of the command group. Note that we need to precede the closing brace with a ; if both { and } appear in the same line.

An often-used sequence used by many shell scripts checks the number of arguments supplied with the command, and then terminates the script with **exit** if the test fails. For instance, a sequence like this:

```
if [ $# -ne 3 ] ; then
   echo "You have not keyed in 3 arguments"
   exit 3
fi
```

can be easily replaced with this sequence using curly braces:

```
[ $# -ne 3 ] && { echo "You have not keyed in 3 arguments" ; exit 3 ; }
```

Why can't we use ( ) instead of { } here? The **exit** statement can terminate a script only if it runs in the same shell that's running the script. This is the case when **exit** runs inside the { }, but not when it runs inside ( ). An **exit** inside ( ) will stop executing the remaining statements in the group, but it doesn't automatically terminate a script.

24.3 export: EXPORTING SHELL VARIABLES

By default, the values stored in shell variables are local to the shell, i.e., they are available only in the shell in which they are defined. They are not passed on to a child shell. But the shell can also *export* these variables (with the **export** statement) recursively to all child processes so that, once defined, they are available globally. You have used this statement before, but now you should understand why you have done so.

Consider a simple script which displays the value of a variable x, assigns a new value to it and then displays the new value again:

```
$ cat var.sh
echo The value of x is $x
x=20                              # Now change the value of x
echo The new value of x is $x
```

First assign 10 to x at the prompt and then execute the script:

```
$ x=10 ; var.sh
The value of x is
The new value of x is 20
$ echo $x
10
```

Value of x not visible in a sub-shell

Value set inside the script doesn't affect value outside script

Because x is a *local* variable in the login shell, its value can't be accessed by **echo** in the script, which is run in a sub-shell. To make x available globally, you need to use the **export** statement before the script is executed:

```
$ x=10 ; export x
$ var.sh
The value of x is 10
The new value of x is 20
$ echo $x
10
```

Value in parent shell now visible here

Value reset inside script (child shell) is not available outside it (parent shell)

When x is exported, its assigned value (10) is available in the script. But when you export a variable it has another important consequence; a reassignment (x=20) made in the script, i.e., a sub-shell, is not seen in the parent shell which executed the script.

You must export the variables you define unless you have strong reasons not to let sub-shells inherit their values. To know whether you have already done so, use **export** without arguments. It lists all environment variables (which are already exported) and user-defined variables (like x) that you have exported. The **env** command also lists exported variables.

Note A variable defined in a process is only local to the process in which it is defined, and is not available in a child process. But when it is exported, it is available recursively to all child processes. However, when the child alters the value of the variable, the change is not seen in the parent.

24.4 RUNNING SCRIPT IN THE CURRENT SHELL: THE . COMMAND

Variable assignments made in the shell's startup file (`.profile` or `.bash_profile`) are *always* seen in the login shell. It's obvious that the profile is executed by the login shell without creating a sub-shell, but how? If you thought that the `.profile` was executed by grouping it with the curly braces

```
{ .profile ; }
```

then you'll be disappointed to see that there is no executable permission for the file:

```
$ ls -l .profile
-rw-r--r--   1 kumar    group         727 Feb 27 23:02 .profile
```

There's a special command which is used to execute any shell script without creating a sub-shell—the . (dot) command. This means that you can make changes to `.profile` and execute it with the . command *without requiring to log out and log in again*:

```
. .profile
```

Many users have the impression that you must log out and log in if you have made any change to the `.profile`. You actually don't need to do that; simply execute the edited file with the . command. You'll need this facility later to execute files containing shell functions.

Note The dot command executes a script without using a sub-shell. It also doesn't require the script to have executable permission.

24.5 let: COMPUTATION—A SECOND LOOK (KORN AND BASH)

Korn and Bash come with a built-in integer handling facility that totally dispenses with the need to use **expr**. You can compute with the **let** statement which is used here both with and without quotes:

```
let sum=256+128                    No whitespace after variable
let sum="256 + 128"                No whitespace after variable
```

If you use whitespace for imparting better readability, then quote the expression. In either case, sum is assigned the result of the expression:

```
$ echo $sum
384
```

Let's see how **let** handles variables. First define three variables; a single **let** does it:

```
$ let x=12 y=18 z=5
$ let z=x+y+$z                     $ not required by let
$ echo $z
35
```

let permits you to get rid of the $ altogether when making an assignment. Since this computational feature is built-in, scripts run much faster than when used with **expr**. Later, we'll be using **let** in place of **expr** in one of our scripts.

A Second Form of Computing with ((and)) The Korn shell and Bash use the (( )) operators that replace the **let** statement itself:

```
$ x=22 y=28 z=5                    Whitespace is unimportant
$ z=$((x+y + z))
$ echo $z
55                                 Can also use z=$((z+=1))
$ z=$((z+1))
$ echo $z
56
```

POSIX recommends the use of ((and)) rather than **let**, and this form is likely to become a standard feature of the shells. It's easier to use too because a variable doesn't have to be preceded by the $. The entire arithmetic operation, however, needs to be preceded by a single $.

24.6 ARRAYS (KORN AND BASH)

Korn and Bash support one-dimensional arrays where the first element has the index 0. Here's how you set and evaluate the value of the third element of the array prompt:

```
$ prompt[2]="Enter your name: "
$ echo ${prompt[2]}
Enter your name:
```

Note that evaluation is done with the curly braces, and prompt[2] is treated just like a variable. It, however, doesn't conflict with a variable prompt that you may also define in the same shell. When a group of elements needs to be assigned, you can use a space-delimited list using either of these two forms:

```
set -A month_arr 0 31 29 31 30 31 30 31 31 30 31 30 31
month_arr=(0 31 29 31 30 31 30 31 31 30 31 30 31)
```

Bash accepts both, but older versions of Korn use only the first form. In either case, the array stores the number of days available in each of the 12 months. The first element had to be deliberately assigned to zero for obvious reasons. Finding out the number of days in June is simple:

```
$ echo ${month_arr[6]}
30
```

Using the @ or * as subscript, you can display all the elements of the array as well as the number of elements. The forms are similar except for the presence of the # in one:

```
$ echo ${month_arr[@]}
0 31 29 31 30 31 30 31 31 30 31 30 31
$ echo ${#month_arr[@]}                    Length of the array
13
```

Can we use arrays to validate an entered date? The next script, **dateval.sh** (Fig. 24.1), does just that. It takes into account the leap year changes (except the one that takes place at the turn of every fourth century).

The first option of the outer **case** construct checks for a null response. The second option uses the expression $n/$n/$n to check for an eight-character string in the form *dd/mm/yy*. Using a changed value of IFS, the components of the date are set to three positional parameters and checked for valid months. The second **case** construct makes the leap year check and then uses an array to validate the day. The **continue** statements take you to loop beginning whenever the test fails the validity check.

Now, let's test the script:

```
$ dateval.sh
Enter a date: [Enter]
No date entered
Enter a date: 28/13/00
Illegal month
Enter a date: 31/04/00
Illegal day
Enter a date: 29/02/01
2001 is not a leap year
Enter a date: 29/02/00
29/02/00 is a valid date
[Ctrl-c]
```

Since the script has no exit path at all, we had to use the interrupt key to terminate execution.

```
#!/usr/bin/ksh
# Script: dateval.sh - Validates a date field using an array
IFS="/"
n="[0-9][0-9]"
set -A month_arr 0 31 29 31 30 31 30 31 31 30 31 30 31
while echo "Enter a date: \c" ; do
  read value
  case "$value" in
       "") echo "No date entered" ; continue ;;
  $n/$n/$n) set $value
            let rem="$3 % 4"
            if [ $2 -gt 12 -o $2 -eq 0 ] ; then
                echo "Illegal month" ; continue
            else
                case "$value" in
         29/02/??) [ $rem -gt 0 ] &&
                   { echo "20$3 is not a leap year" ; continue ; } ;;
                *) [ $1 -gt ${month_arr[$2]} -o $1 -eq 0 ] &&
                   { echo "Illegal day" ; continue ; } ;;
                esac
            fi;;
         *) echo "Invalid date" ; continue ;;
  esac
  echo "$1/$2/$3" is a valid date
done
```

Fig. 24.1 **dateval.sh**

24.7 STRING HANDLING (KORN AND BASH)

Korn and Bash don't need **expr** as they have adequate string handling features themselves. Unlike **expr**, they use wild-cards but not regular expressions. All forms of usage require curly braces to enclose the variable name along with some special symbols. The subtle variations in their forms make them difficult to remember and sometimes uncomfortable to work with.

Length of String The length of a string is easily found by preceding the variable name with a **#**. Consider this example:

```
$ name="vinton cerf"
$ echo ${#name}
11
```

You can now use this expression with an **if** statement to check the length of a string. This built-in feature is not only easier to use than its corresponding **expr** version *(16.9.2)*, but is also comparatively faster:

Korn and Bash

```
if [ ${#name} -gt 20 ] ; then
```

This form should appear familiar to you as **perl** uses a similar form to evaluate the length of an array *(21.10)*.

24.7.1 Extracting a String by Pattern Matching

You can extract a substring using a special pattern matching feature. These functions make use of two characters—# and %. Their selection seems to have been based on mnemonic considerations. # is used to match at the beginning and % at the end, and both are used inside curly braces when evaluating a variable.

To remove the extension from a filename, previously you had to use an external command—**basename** *(16.12.2)*. This time, you can use a variable's ${*variable%pattern*} format to do that:

```
$ filename=quotation.txt
$ echo ${filename%txt}
quotation.                                    txt stripped off
```

The % symbol after the variable name deletes the *shortest* string that matches the variable's contents at the *end*. Had there been two %s instead of one, the expression would have matched the longest one. Let's now use %% with wild-cards to extract the hostname from an FQDN:

```
$ fqdn=java.sun.com
$ echo ${fqdn%%.*}
java
```

You'll recall that **basename** can also extract the base filename from a pathname. This requires you to delete the longest pattern which matches the pattern */, but at the beginning of the variable's value:

```
$ filename="/var/mail/henry"
$ echo ${filename##*/}
henry
```

This deletes the segment, /var/mail—the longest pattern that matches the pattern */ at the beginning. The pattern matching forms of Korn and Bash are listed in Table 24.1.

Table 24.1 Pattern Matching Operators Used by Korn and Bash

Form	*Evaluates to segment remaining after deleting*
${*var#pat*}	Shortest segment that matches *pat* at beginning of $*var*
${*var##pat*}	Longest segment that matches *pat* at beginning of $*var*
${*var%pat*}	Shortest segment that matches *pat* at end of $*var*
${*var%%pat*}	Longest segment that matches *pat* at end of $*var*

24.8 CONDITIONAL PARAMETER SUBSTITUTION

To continue on the subject of variable evaluation, you can evaluate a variable depending on whether it has a null or defined value. This feature is known as *parameter substitution*, and is available in the Bourne shell also. It takes this general form:

${<*var*>:<*opt*><*stg*>}

This time, the variable <*var*> is followed by a colon and any of the symbols +, -, = or ? as <*opt*>. The symbol is followed by the string <*stg*>. In all cases barring one, this doesn't alter the value of the variable, but only determines the way it is *evaluated*. This evaluation can be done in four ways.

The + Option Here, *var* evaluates to *stg* if it is defined and assigned a nonnull string. This feature can be used to set a variable to the output of a command, and echo a message if the variable is nonnull:

```
found=`ls`
echo ${found:+"This directory is not empty"}
```

`ls` displays nothing if it finds no files, in which case the variable found is set to a null string. However, the message is echoed if **ls** finds at least one file.

The - Option Here, *var* is evaluated to *stg* if it is undefined or assigned a null string (the opposite of the + option). You can use this feature in a program which prompts for a filename, and then uses a default value when the user simply presses *[Enter]*:

```
echo "Enter the filename : \c"
read flname
fname=${flname:-emp.lst}                    Instead of using if [ -z $flname ]
```

If `flname` is null or is not set, it *evaluates* to the string `emp.lst`. The value of `flname`, however, still remains null. This compact assignment dispenses with the need for an **if** conditional.

The = Option This also works similarly except that it goes a step further and *makes the assignment* to the variable that is evaluated. With the = option, you can use parameter substitution with a command without making the intermediate assignment:

```
echo "Enter the filename : \c"
read flname
grep $pattern ${flname:=emp.lst}            flname is now assigned
```

Note that in the last statement, the variable `flname` itself got assigned (provided it was either unset or set to a null string). This feature is most useful in initializing a loop which iterates as long as a counter inside the loop matches the control command in the **while** command line. The following statements

```
x=1 ; while [ $x -le 10 ]
```

can now be combined in one:

```
while [ ${x:=1} -le 10 ]
```

The ? Option It evaluates the parameter if the variable is assigned and nonnull, otherwise it echoes the string following it. Additionally, the shell is killed. This is quite useful in terminating a script if the user fails to respond properly to shell directives:

```
echo "Enter the filename : \c"
read flname
grep $pattern ${flname:?"No filename entered"}
```

If no filename is entered here, the message `No filename entered` is displayed. The script is also aborted without the use of an explicit **exit** command.

Apart from the = operator, the other operators can also be used with positional parameters. You can now easily set a variable to some default if the script is invoked without an argument:

```
flname=${1:-emp.lst}                    $1 is null if script invoked without arguments
```

You can now compress some of the earlier scripts to even shorter command sequences. You'll see some of these applications later in the chapter.

Note Only the = option actually assigns a value to a variable; the others merely control the way the variable is evaluated.

24.9 MERGING STREAMS

You have seen the utility of redirecting a loop at the **done** keyword *(16.11)*. This also meant redirecting individual statements inside the loop to /dev/tty when these statements needed to send output to the terminal. When there are a large number of such terminal-destined statements, separately using >/dev/tty with each one of them can be quite tedious. The shell offers a simple solution to this problem.

Though the standard output and standard error are two separate streams, the shell lets you merge the two so that they can be collectively manipulated. Once you do that, you effectively have a single stream, which you can subsequently use with any of the shell's redirection and piping symbols. This is done with the & operator following the redirection symbol. When the **echo** statement is placed in a script in this way:

```
echo "None of the patterns found" 1>&2
```

then, irrespective of the destination of the rest of the script, the output of this statement will always come to the destination of the standard error. It's like saying: "Send the standard output to the destination of the standard error". Since the default file descriptor for standard output is 1, you can also use >&2.

The notation 1>&2 merges the standard output with the standard error, which, in the absence of further redirection, is connected to the terminal. However, when you redirect the standard error of the entire script in this way:

```
emp8.sh > sclist 2> errlist
```

all script statements having the symbols 1>&2 affixed, will actually write to `errlist`. The rest of the script output will, however, be saved in `sclist`.

For the reverse situation, i.e., sending the standard error to the destination of the standard output, the notation becomes 2>&1. You may require this to be done on many occasions, especially when a job is run in the background, and you still want the error messages to be held in the same file that contains the actual output.

24.10 SHELL FUNCTIONS

A *shell function* is like any other function; it executes a group of statements as a bunch. Optionally, it also returns a value. This construct is available in most modern shells where you can use them to condense important routines to short sequences. The syntax is simple:

```
function_name() {
    statements
    return value                                    Optional
}
```

The function definition is followed by (), and the body is enclosed within curly braces. On invocation, the function executes all *statements* in the body. The **return** statement, when present, returns a value representing the success or failure of the function (and not a string value).

Let's consider a simple application. When viewing the listing of a large number of files in a directory, you are often compelled to use **ls -l | more**. This command sequence is an ideal candidate for a shell function, which we'll name 11:

```
$ 11() {                                    Function defined in the command line
> ls -l $* | more                           is available in current shell only
> }
```

Like shell scripts, shell functions also use command line arguments (like $1, $2, etc.). $*, "$@" and $# also retain their usual significance in functions. Even though you need the () in the function definition, you must not use them when invoking the function. You can now invoke the function with or without arguments:

```
11                                          Executes ls -l | more
11 ux3rd??                                  Executes ls -l ux3rd?? | more
```

We have used the **11** function more like a procedure, but a shell function can also return its exit status with the **return** statement. This exit status can be gathered from the shell parameter, $?.

Shell functions can be defined at a number of places:

- At the beginning of every script using them. Since shell statements are executed in the interpretive mode, a shell function must precede the statements that call it.
- In the `.profile` so that they are available in the current session.
- In a separate "library" file so that other applications can also use them.

Since the above function is often required, it's better you create a file **mainfunc.sh** to hold it. You then have to place the following entry at the beginning of every shell script that needs to use it:

```
. mainfunc.sh                               Functions available in the current shell
```

Caution The positional parameters made available to shell scripts externally are not available *directly* to a shell function. To make them available, store these parameters in shell variables first.

24.10.1 Generating a Filename from the System Date

As a system administrator, you'll often require to maintain separate files for each day of a specific activity. These filenames can be derived from the system date, so you can easily identify a file pertaining to a certain day. It's a good idea to have this value echoed by a shell function **dated_fname**, which in turn gets it from the **date** output:

```
dated_fname() {
   set -- `date`
   year=`expr $6 : '..\(..\)'`                    Picks up last two characters from year
   echo "$2$3_$year"
}
```

When you invoke the function, **echo** displays a string derived from the current date:

```
$ dated_fname
Jan28_03
```

This string can be used to frame filenames that are needed not more than once a day. Oracle users can easily use this function to have a system-generated dump filename for the **exp** command:

```
exp scott/tiger file=`dated_fname`
```

This will generate the export dump file `Jan28_03.dmp`. Place this function definition in the library file **mainfunc.sh** that we created previously. We'll use this file later—after we have added two more functions to it.

24.10.2 To Continue or Not to Continue

We'll now consider a shell function which actually returns a value. Recall the sequence used at the end of the script, **dentry1.sh** *(16.17.2)*, which prompts the user for a `y` to continue, or an `n` to terminate the outermost loop. This routine is very often used inside shell scripts, and it's a good idea to convert this into a function, **anymore**:

```
anymore() {
    echo "\n$1 ?(y/n) : \c" 1>&2                  Prompt supplied as argument
    read response
    case "$response" in
      y|Y) echo  1>&2 ; return 0 ;;
        *) return 1 ;;
    esac
}
```

The function uses its argument `$1` to determine what the entire prompt will look like. When this function is invoked with the string `Wish to continue`, you are prompted for a response:

```
$ anymore "Wish to continue"

Wish to continue ?(y/n) : n
$ echo $?
1
```

Same value as specified in `return` *statement*

We'll make use of the return value of this function later in the chapter.

24.10.3 Validating Data Entry

Consider the script, **dentry1.sh** *(16.17.2)*, where you used a **while** loop to repeatedly prompt the user for a valid entry. This sequence can also be easily converted into a shell function, **valid_string**. We expect the function to check two things—first that something is entered at all, and whether it exceeds a certain length:

```
valid_string() {
   while echo "$1 \c"  1>&2 ; do
      read name
      case $name in
         "") echo "Nothing entered"  1>&2 ; continue ;;
          *) if [ `expr "$name" : '.*'` -gt $2 ] ; then
                echo "Maximum $2 characters permitted" 1>&2
             else
                break
             fi ;;
      esac
   done
   echo $name
}
```

The function takes two arguments—the prompt string and the maximum number of characters permitted. We'll place this function in our library file, **mainfunc.sh**, and call it from the script **user_passwd.sh** (Fig. 24.2).

```
#!/bin/sh
# Script: user_passwd.sh - Uses a shell function
. mainfunc.sh                          # Script containing valid_string function

user=`valid_string "Enter your user-id :" 16`
stty -echo                             # Password not to be echoed
password=`valid_string "Enter your password:" 5`
stty echo                              # Turns on echoing facility
echo "\nYour user-id is $user and your password is $password"
```

Fig. 24.2 user_passwd.sh

This small script accepts the user's name and password, which can't exceed 16 and 5 characters in length, respectively. A sample session shows how shell functions can reduce script size:

```
$ user_passwd.sh
Enter your user-id : robert louis stevenson
Maximum 16 characters permitted
Enter your user-id : scott
Enter your password:
Nothing entered
Enter your password: *****                    Doesn't show on screen
Your user-id is scott and your password is tiger
```

Invocation of a shell function reduces disk I/O since the function is resident in memory. On the other hand, when you execute a shell script, the shell first scans the disk for the script file. However, you must take care to see that you don't keep a lot of functions in memory as that would eat up CPU resources. Shell functions are better than aliases in every way, and they work in the Bourne shell also.

Tip If a number of shell functions are used by multiple programs, you should place all of them in a single "library" file, and store the file at a convenient location. At the beginning of every script which requires these functions, insert a statement that executes the library file with the dot command.

24.11 eval: EVALUATING TWICE

The previous script requires that you set the variable names ($user and $password) for reading each input. We would prefer to have a more general script where the variable name itself can be generated by the script on the fly. We would store the prompts as variables and read the input as "numbered variables".

First define a couple of prompts at the prompt and then attempt to issue the first prompt with a numbered variable:

```
$ prompt1="Employee id : " ; prompt2="Name : " ; prompt3="Designation : "
$ x=1
$ echo $prompt$x
1
```

Instead of echoing `Employee id :` , it simply echoes the value 1, viz. the value of x. This happens because the shell evaluates the command line from left to right. It first encounters the $ and evaluates $prompt; obviously $prompt is undefined. The shell then evaluates $x, which has the value 1.

The shell's **eval** statement evaluates a command line twice. The first pass ignores any metacharacter escaped with a \. The second pass ignores the \ and evaluates the metacharacter normally. This is exactly what we require the shell to do in our previous example. If we escape the first $ (like \$prompt$x), the first pass will evaluate $x first so that we have prompt1, and the second pass then evaluates $prompt1. This is done by prefixing the **echo** command with **eval**:

```
$ x=1 ; eval echo \$prompt$x
Employee id :
```

We have a numbered prompt; we now need to use a numbered variable to read the input. For instance, the responses against numbered prompts can also be held in variables value1, value2, value3, etc. The following sequence does it for the second prompt:

```
$ { x=2
> eval echo \$prompt$x '\\c'
> read value$x
> eval echo \$value$x
> }
Name : rahul verma
rahul verma
```

This has no problems

Let's consider another application of **eval**. We can access a positional parameter with $1 or whatever, but can we access the last parameter directly? Since we have the value of $# available, we can use the services of **eval**:

```
$ tail -1 /etc/passwd
martha:x:605:100:martha mitchell:/home/martha:/bin/ksh
$ IFS=:
$ set `tail -1 /etc/passwd`     set -- not required here
$ eval echo \$$#
/bin/ksh
```

Look, using **eval**, we don't even have to know the number of fields in a line of /etc/passwd. In the next section, we'll use the services of **eval** both to generate prompts as well as to read user input.

24.11.1 A Generalized Data Entry Script with eval

We'll now use the **eval** feature to accept inputs into six fields with a generalized script, **dentry2.sh** (Fig. 24.3). There are six prompts defined at the beginning of the script as numbered variables, and an inner **while** loop issues all of them in turn. The values are read into the variables value1, value2, etc., and appended to the variable $rekord, using the | as delimiter each time.

```
#!/bin/sh
# Script: dentry2.sh - Uses eval and shell functions

trap 'echo "Program interrupted"; exit' HUP INT TERM
. mainfunc.sh        # Invokes functions valid_string() and anymore ()

prompt1="Employee id : " ; prompt2="Name : " ; prompt3="Designation : "
prompt4="Department : " ; prompt5="Date birth : " ; prompt6="Basic pay : "
rekord=

flname=`valid_string "Enter the output filename: " 8`
while true ; do
    while [ ${x:=1} -le 6 ] ; do                    # x first set to 1
        eval echo \$prompt$x '\\c' 1>&2
        read value$x
        rekord="${rekord}`eval echo \\$value$x`|"
        x=`expr $x + 1`
    done
    echo "$rekord"
    anymore "More entries to add" 1>&2 || break
done > $flname
```

Fig. 24.3 **dentry2.sh**

Note how the use of shell functions has kept the script size small. Here's a sample interaction:

```
$ dentry2.sh
Enter the output filename : newlist

Employee id : 2244
Name : prakash kumar
Designation : director
Department : marketing
Date birth : 27/02/44
Basic pay : 6700

More entries to add ?(y/n) : y

Employee id : 4789
Name : v.k. singh
Designation : manager
Department : personnel
Date birth : 21/12/46
Basic pay : 6500

More entries to add ?(y/n) : n
```

When you see `newlist`, you will find two records with an extra | at the end of each:

```
$ cat newlist
2244|prakash kumar|director|marketing|27/02/44|6700|
4789|v.k. singh|manager|personnel|21/12/46|6500|
```

Now, this is amazing; thanks to **eval**, we have managed to read six user responses into six variables with a minuscule script!

24.12 THE exec STATEMENT

Your study of the mechanism of process creation *(10.4)* led you to the **exec** system call—one that overlays a forked process. This property has some importance to shell scripters who sometimes need to overwrite the current shell itself with another program's code. This is something we haven't done yet, but if you precede any UNIX command with **exec**, the command overwrites the current process—often the shell. This has the effect of logging you out after the completion of the command:

```
$ exec date
Tue Jan 28 21:21:52 IST 2003
login:
```

Shell no longer exists!

Sometimes, you might want to let a user run a single program automatically on logging in and deny her an escape to the shell. You can place the command in the `.profile`, duly preceded by **exec**. The shell overlays itself with the code of the program to be executed, and when command execution is complete, the user is logged out (since there's no shell waiting for it).

24.12.1 Using exec to Create Additional File Descriptors

exec has another important property; it can redirect the standard streams for an entire script. If a script has several commands whose standard output go to a single file, then instead of using separate redirection symbols for each, you can use **exec** to reassign their default destination like this:

```
exec > foundfile
```

Can use >> also

What's the big deal you might say; one could redirect the script itself. But **exec** can create several streams apart from the standard three (0, 1 and 2), each with its own file descriptor. For instance, you can create a file descriptor 3 for directing all output and associate it with a physical file `foundfile`:

```
exec 3>invalidfile
```

In system call parlance *(25.2.1)*, you have *opened* this file to generate the file descriptor 3. Subsequent access to this file can be made by using this file descriptor. You can now write the file by merging the standard output stream with the file descriptor 3:

```
echo "This goes to invalidfile" 1>&3
```

With this powerful I/O handler, you should now be able to handle files in a simpler and more elegant way. Let's design a script which reads emp-ids from a file. It then searches `emp.lst` and saves in three separate files the following:

- The lines found.
- The emp-ids not found.
- Badly formed emp-ids.

First, here's the file that contains the emp-ids. It contains two three-digit emp-ids which should be trapped by the script:

```
$ cat empid.lst
2233
9765
2476
789
3564
9877
0110
245
2954
```

The script, **countpat.sh** (Fig. 24.4), divides the standard output into three streams and redirects them to three separate files. It requires four arguments—the file containing the patterns and the files for the three streams.

The standard output streams are merged with the file descriptors 1, 3 and 4 using **exec**. Note that we have also set **$1** as the source of all standard input. This means that the **read** statement in the loop will take input from **$1**—the file containing the patterns. Once all file writing is over, the standard output stream has to be reassigned to the terminal (**exec >/dev/tty**), otherwise the message Job Over will also be saved in the filename passed to **$2**.

This script is quite clean and has two statements using the merging symbols. The **grep** statement uses the standard output's file descriptor, so no merging is required. The script takes four arguments and diverts the output into three of them:

```
#!/bin/sh
# Script: countpat.sh -- Uses exec to handle multiple files
#
exec > $2                          # Open file 1 for storing selected lines
exec 3> $3                         # Open file 4 for storing patterns not found
exec 4> $4                         # Open file 5 for storing invalid patterns
[ $# -ne 4 ] && { echo "4 arguments required" ; exit 2 ; }
exec < $1                                      # Redirecting input
while read pattern ; do
     case "$pattern" in
        ????) grep $pattern emp.lst ||
          echo $pattern not found in file 1>&3 ;;
           *) echo $pattern not a four-character string 1>&4 ;;
     esac
done
exec >/dev/tty                 # Redirects standard output back to terminal
echo Job Over
```

Fig. 24.4 **countpat.sh**

```
$ countpat.sh empid.lst foundfile notfoundfile invalidfile
Job Over
```

The message appears on the terminal instead of going to any of these files. Now, just have a look at the three files and see for yourself what has actually happened:

```
$ cat foundfile
2233|a.k. shukla      |g.m.      |sales     |12/12/52|6000
2476|anil aggarwal    |manager   |sales     |05/01/59|5000
3564|sudhir Agarwal   |executive|personnel |07/06/47|7500
0110|v.k. agrawal     |g.m.      |marketing |12/31/40|9000
$ cat notfoundfile
9765 not found in file
9877 not found in file
2954 not found in file
$ cat invalidfile
789 not a four-character string
245 not a four-character string
```

This then is the power of **exec**. It can open several files together, and access each one separately in the same way **perl** uses its own filehandles and system calls use file descriptors. It is always preferable to use file descriptors instead of filenames because using **exec**, you can keep them open, and this makes I/O operations efficient.

24.13 CONCLUSION

We examined the shell's environment and discussed some of the useful features of the Korn and Bash shells. We also discussed two advanced features, **eval** and **exec**, that are available in the Bourne shell also. Like **sed**, it takes time to master them, but they are extremely useful tools. The **eval** feature is indeed unique, but knowing **exec** will help you understand how file I/O actually takes place. In the next chapter, we take up this subject in our study of the UNIX system calls.