Unix Shell Programming

3rd Edition
Covers bash 3.0

Learning the bash *Shell*

O'REILLY®

Cameron Newham

CHAPTER 4

Basic Shell Programming

If you have become familiar with the customization techniques we presented in the previous chapter, you have probably run into various modifications to your environment that you want to make but can't—yet. Shell programming makes these possible.

bash has some of the most advanced programming capabilities of any command interpreter of its type. Although its syntax is nowhere near as elegant or consistent as that of most conventional programming languages, its power and flexibility are comparable. In fact, *bash* can be used as a complete environment for writing software prototypes.

Some aspects of *bash* programming are really extensions of the customization techniques we have already seen, while others resemble traditional programming language features. We have structured this chapter so that if you aren't a programmer, you can read this chapter and do quite a bit more than you could with the information in the previous chapter. Experience with a conventional programming language like Pascal or C is helpful (though not strictly necessary) for subsequent chapters. Throughout the rest of the book, we will encounter occasional programming problems, called *tasks*, whose solutions make use of the concepts we cover.

Shell Scripts and Functions

A *script* (a file that contains shell commands) is a shell program. Your *.bash_profile* and environment files, discussed in the previous chapter, are shell scripts.

You can create a script using the text editor of your choice. Once you have created one, there are two ways to run it. One, which we have already covered, is to type **source** *scriptname*. This causes the commands in the script to be read and run as if you typed them in.

The second way to run a script is simply to type its name and hit RETURN, just as if you were invoking a built-in command. This, of course, is the more convenient way. This method makes the script look just like any other UNIX command, and in fact

several "regular" commands are implemented as shell scripts (i.e., not as programs originally written in C or some other language), including *spell*, *man* on some systems, and various commands for system administrators. The resulting lack of distinction between "user command files" and "built-in commands" is one factor in UNIX's extensibility and, hence, its favored status among programmers.

You can run a script by typing its name only if the directory where the script is located is in your command search path, or . (the current directory) is part of your command search path, i.e., the script's directory path (as discussed in Chapter 3). If these aren't in your path, you must type *./scriptname*, which is really the same thing as typing the script's absolute pathname (see Chapter 1).

Before you can invoke the shell script by name, you must also give it "execute" permission. If you are familiar with the UNIX filesystem, you know that files have three types of permissions (read, write, and execute) and that those permissions apply to three categories of user (the file's owner, a *group* of users, and everyone else). Normally, when you create a file with a text editor, the file is set up with read and write permission for you and read-only permission for everyone else.

Therefore you must give your script execute permission explicitly, by using the *chmod* command. The simplest way to do this is to type:

```
$ chmod +x scriptname
```

Your text editor will preserve this permission if you make subsequent changes to your script. If you don't add execute permission to the script and you try to invoke it, the shell will print the message:

```
scriptname: Permission denied
```

But there is a more important difference between the two ways of running shell scripts. While using **source** causes the commands in the script to be run as if they were part of your login session, the "just the name" method causes the shell to do a series of things. First, it runs another copy of the shell as a subprocess; this is called a *subshell*. The subshell then takes commands from the script, runs them, and terminates, handing control back to the parent shell.

Figure 4-1 shows how the shell executes scripts. Assume you have a simple shell script called *alice* that contains the commands *hatter* and *gryphon*. In .a, typing **source alice** causes the two commands to run in the same shell, just as if you had typed them in by hand. .b shows what happens when you type just **alice**: the commands run in the subshell while the parent shell waits for the subshell to finish.

You may find it interesting to compare this with the situation in .c, which shows what happens when you type **alice &**. As you will recall from Chapter 1, the **&** makes the command run in the *background*, which is really just another term for "subprocess." It turns out that the only significant difference between .c and .b is that you have control of your terminal or workstation while the command runs—you need not wait until it finishes before you can enter further commands.

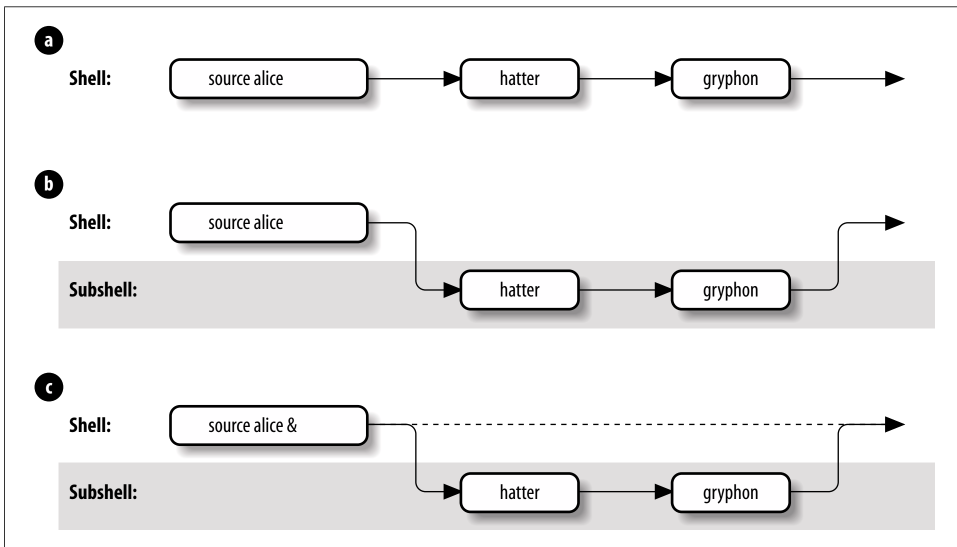

Figure 4-1. Ways to run a shell script

There are many ramifications to using subshells. An important one is that the *export*ed environment variables that we saw in the last chapter (e.g., **TERM**, **EDITOR**, **PWD**) are known in subshells, whereas other shell variables (such as any that you define in your *.bash_profile* without an *export* statement) are not.

Other issues involving subshells are too complex to go into now; see Chapter 7 and Chapter 8 for more details about subshell I/O and process characteristics, respectively. For now, just bear in mind that a script normally runs in a subshell.

Functions

bash's *function* feature is an expanded version of a similar facility in the System V Bourne shell and a few other shells. A function is sort of a script-within-a-script; you use it to define some shell code by name and store it in the shell's memory, to be invoked and run later.

Functions improve the shell's programmability significantly, for two main reasons. First, when you invoke a function, it is already in the shell's memory; therefore a function runs faster. Modern computers have plenty of memory, so there is no need to worry about the amount of space a typical function takes up. For this reason, most people define as many commonly used functions as possible rather than keep lots of scripts around.

The other advantage of functions is that they are ideal for organizing long shell scripts into modular "chunks" of code that are easier to develop and maintain. If you aren't a programmer, ask one what life would be like without functions (also called *procedures* or *subroutines* in other languages) and you'll probably get an earful.

To define a function, you can use either one of two forms:

```
function functname{
    shell commands}
```

or:

```
functname ()
{
    shell commands}
```

There is no functional difference between the two. We will use both forms in this book. You can also delete a function definition with the command **unset -f** *functname*.

When you define a function, you tell the shell to store its name and definition (i.e., the shell commands it contains) in memory. If you want to run the function later, just type in its name followed by any arguments, as if it were a shell script.

You can find out what functions are defined in your login session by typing **declare -f**. The shell will print not just the names but the definitions of all functions, in alphabetical order by function name. Since this may result in long output, you might want to pipe the output through *more* or redirect it to a file for examination with a text editor. If you just want to see the names of the functions, you can use **declare -F**.* We will look at **declare** in more detail in Chapter 6.

Apart from the advantages, there are two important differences between functions and scripts. First, functions do not run in separate processes, as scripts do when you invoke them by name; the "semantics" of running a function are more like those of your *.bash_profile* when you log in or any script when invoked with the **source** command. Second, if a function has the same name as a script or executable program, the function takes precedence.

This is a good time to show the order of precedence for the various sources of commands when you type a command to the shell:

1. Aliases
2. Keywords such as **function** and several others, like **if** and **for**, which we will see in Chapter 5
3. Functions
4. Built-ins like **cd** and **type**
5. Scripts and executable programs, for which the shell searches in the directories listed in the **PATH** environment variable

Thus, an alias takes precedence over a function or a script with the same name. You can, however, change the order of precedence by using the built-ins **command**, **builtin**, and **enable**. This allows you to define functions, aliases, and script files with

* The **-F** option is not available in versions of *bash* prior to 2.0.

the same names, and select which one you want to execute. We'll examine this process in more detail in the section on command-line processing in Chapter 7.

If you need to know the exact source of a command, there are options to the **type** built-in command that we saw in Chapter 3. **type** by itself will print how *bash* would interpret the command, based on the search locations listed above. If you supply more than one argument to **type**, it will print the information for each command in turn. If you had a shell script, a function, and an alias all called *dodo*, **type** would tell you that *dodo*, as an alias, would be used if you typed *dodo*.

type has several options that allow you to find specific details of a command. If you want to find out all of the definitions for *dodo* you can use **type -a**. This will produce output similar to the following:

```
$ type -all dodo
dodo is aliased to `echo "Everybody has won, and all must have prizes"'
dodo is a function
dodo ()
{
    echo "Everybody has won, and all must have prizes"
}
dodo is ./dodo
```

It is also possible to restrict the search to commands that are executable files or shell scripts by using the **-p** option. If the command as typed to *bash* executes a file or shell script, the path name of the file is returned; otherwise, nothing is printed.

The -**P** option forces type to look for executable files or shell scripts even if the result of -**t** would not return *file*.

A further option, **-f**, suppresses shell function lookup, i.e., only keywords, files and aliases will be returned.*

The default output from **type** is verbose; it will give you the full definition for an alias or function. By using the **-t** option, you can restrict this to a single word descriptor: *alias*, *keyword*, *function*, *builtin*, or *file*. For example:

```
$ type -t bash
file
$ type -t if
keyword
```

The **-t** option can also be used with all other options.

We will refer mainly to scripts throughout the remainder of this book, but unless we note otherwise, you should assume that whatever we say applies equally to functions.

* The options -**f** and -**P** are not available in versions of *bash* prior to 2.05b.

Shell Variables

bash derives much of its programming functionality from shell variables. We've already seen the basics of variables. To recap briefly: they are named places to store data, usually in the form of character strings, and their values can be obtained by preceding their names with dollar signs ($). Certain variables, called *environment variables*, are conventionally named in all capital letters, and their values are made known (with the **export** statement) to subprocesses.

If you are a programmer, you already know that just about every major programming language uses variables in some way; in fact, an important way of characterizing differences between languages is comparing their facilities for variables.

The chief difference between *bash*'s variable schema and those of conventional languages is that *bash*'s places heavy emphasis on character strings. (Thus it has more in common with a special-purpose language like SNOBOL than a general-purpose one like Pascal.) This is also true of the Bourne shell and the C shell, but *bash* goes beyond them by having additional mechanisms for handling integers explicitly.

Positional Parameters

As we have already seen, you can define values for variables with statements of the form **varname=value**, e.g.:

```
$ hatter=mad
$ echo "$hatter"
mad
```

The shell predefines some environment variables when you log in. There are other built-in variables that are vital to shell programming. We will look at a few of them now and save the others for later.

The most important special, built-in variables are called *positional parameters*. These hold the command-line arguments to scripts when they are invoked. Positional parameters have the names **1**, **2**, **3**, etc., meaning that their values are denoted by **$1**, **$2**, **$3**, etc. There is also a positional parameter **0**, whose value is the name of the script (i.e., the command typed in to invoke it).

Two special variables contain all of the positional parameters (except positional parameter **0**): * and @. The difference between them is subtle but important, and it's apparent only when they are within double quotes.

"**$***" is a single string that consists of all of the positional parameters, separated by the first character in the value of the environment variable **IFS** (internal field separator), which is a space, TAB, and NEWLINE by default. On the other hand, "**$@**" is equal to "**$1**" "**$2**"... "**$*N***", where *N* is the number of positional parameters. That is, it's equal to *N* separate double-quoted strings, which are separated by spaces. If there are no positional parameters, "**$@**" expands to nothing. We'll explore the ramifications of this difference in a little while.

The variable # holds the number of positional parameters (as a character string). All of these variables are "read-only," meaning that you can't assign new values to them within scripts.

For example, assume that you have the following simple shell script:

```
echo "alice: $@"
echo "$0: $1 $2 $3 $4"
echo "$# arguments"
```

Assume further that the script is called *alice*. Then if you type **alice in wonderland**, you will see the following output:

```
alice: in wonderland
alice: in wonderland
2 arguments
```

In this case, **$3** and **$4** are unset, which means that the shell will substitute the empty (or null) string for them.*

Positional parameters in functions

Shell functions use positional parameters and special variables like * and # in exactly the same way as shell scripts do. If you wanted to define *alice* as a function, you could put the following in your *.bash_profile* or environment file:

```
function alice
{
    echo "alice: $*"
    echo "$0: $1 $2 $3 $4"
    echo "$# arguments"
}
```

You will get the same result if you type **alice in wonderland**.

Typically, several shell functions are defined within a single shell script. Therefore each function will need to handle its own arguments, which in turn means that each function needs to keep track of positional parameters separately. Sure enough, each function has its own copies of these variables (even though functions don't run in their own subshells, as scripts do); we say that such variables are *local* to the function.

However, other variables defined within functions are not local (they are *global*), meaning that their values are known throughout the entire shell script. For example, assume that you have a shell script called *ascript* that contains this:

```
function afunc
{
  echo in function: $0 $1 $2
  var1="in function"
  echo var1: $var1
}
```

* Unless the option **nounset** is turned on, in which case the shell will return an error message.

```
var1="outside function"
echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

If you invoke this script by typing **ascript arg1 arg2**, you will see this output:

```
var1: outside function
ascript: arg1 arg2
in function: ascript funcarg1 funcarg2
var1: in function
var1: in function
ascript: arg1 arg2
```

In other words, the function *afunc* changes the value of the variable **var1** from "outside function" to "in function," and that change is known outside the function, while **$1** and **$2** have different values in the function and the main script. Notice that **$0** *doesn't* change because the function executes in the environment of the shell script and **$0** takes the name of the script. Figure 4-2 shows the scope of each variable graphically.

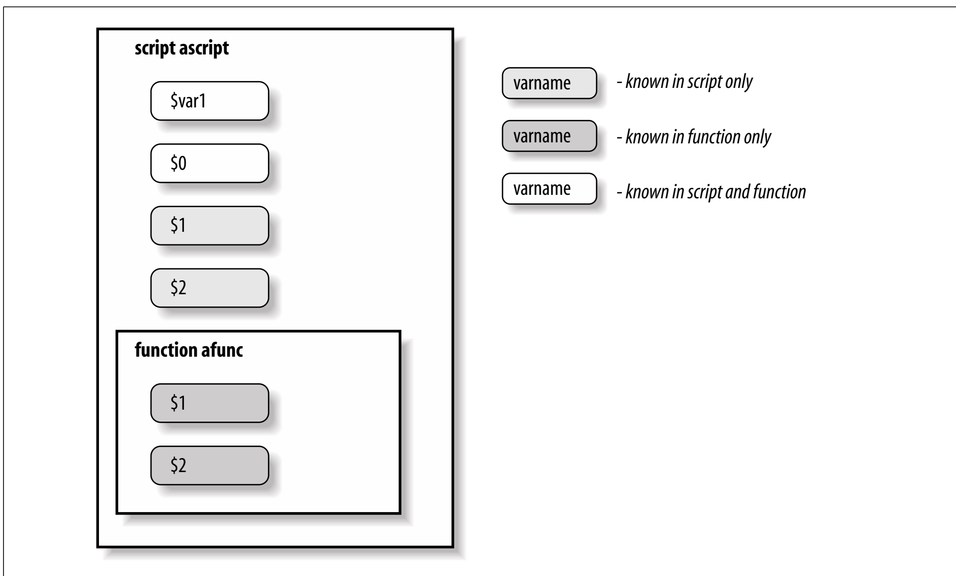

Figure 4-2. Functions have their own positional parameters

Local Variables in Functions

A **local** statement inside a function definition makes the variables involved all become *local* to that function. The ability to define variables that are local to "subprogram" units (procedures, functions, subroutines, etc.) is necessary for writing

large programs, because it helps keep subprograms independent of the main program and of each other.

Here is the function from our last example with the variable *var1* made local:

```
function afunc
{
  local var1
  echo in function: $0 $1 $2

  var1="in function"
  echo var1: $var1
}
```

Now the result of running **ascript arg1 arg2** is:

```
var1: outside function
ascript: arg1 arg2
in function: ascript funcarg1 funcarg2
var1: in function
var1: outside function
ascript: arg1 arg2
```

Figure 4-3 shows the scope of each variable in our new script. Note that **afunc** now has its own, local copy of *var1*, although the original *var1* would still be used by any other functions that **ascript** invokes.

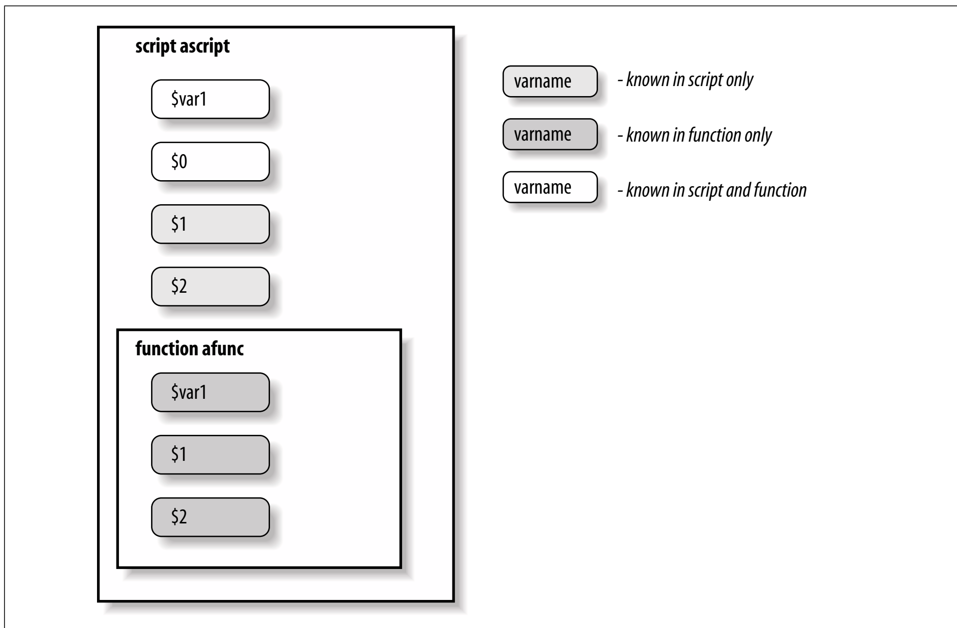

Figure 4-3. Functions can have local variables

Quoting with $@ and $*

Now that we have this background, let's take a closer look at "$@" and "$*". These variables are two of the shell's greatest idiosyncracies, so we'll discuss some of the most common sources of confusion.

- Why are the elements of "$*" separated by the first character of **IFS** instead of just spaces? To give you output flexibility. As a simple example, let's say you want to print a list of positional parameters separated by commas. This script would do it:

  ```
  IFS=,
  echo "$*"
  ```

  Changing **IFS** in a script is risky, but it's probably OK as long as nothing else in the script depends on it. If this script were called *arglist*, then the command **arglist alice dormouse hatter** would produce the output **alice,dormouse,hatter**. Chapter 5 and Chapter 10 contain other examples of changing **IFS**.

- Why does "$@" act like N separate double-quoted strings? To allow you to use them again as separate values. For example, say you want to call a function within your script with the same list of positional parameters, like this:

  ```
  function countargs
  {
      echo "$# args."
  }
  ```

  Assume your script is called with the same arguments as *arglist* above. Then if it contains the command **countargs** "$*", the function will print **1 args**. But if the command is **countargs** "$@", the function will print **3 args**.

More on Variable Syntax

Before we show the many things you can do with shell variables, we have to point out a simplification we have been making: the syntax of ***$varname*** for taking the value of a variable is actually the simple form of the more general syntax, ***${varname}***.

Why two syntaxes? For one thing, the more general syntax is necessary if your code refers to more than nine positional parameters: you must use **${10}** for the tenth instead of **$10**. Aside from that, consider the following case where you would like to place an underscore after your user ID:

```
echo $UID_
```

The shell will try to use **UID_** as the name of the variable. Unless, by chance, **$UID_** already exists, this won't print anything (the value being *null* or the empty string, ""). To obtain the desired result, you need to enclose the shell variable in curly brackets:

```
echo ${UID}_
```

It is safe to omit the curly brackets ({}) if the variable name is followed by a character that isn't a letter, digit, or underscore.

String Operators

The curly-bracket syntax allows for the shell's *string operators*. String operators allow you to manipulate values of variables in various useful ways without having to write full-blown programs or resort to external UNIX utilities. You can do a lot with string-handling operators even if you haven't yet mastered the programming features we'll see in later chapters.

In particular, string operators let you do the following:

- Ensure that variables exist (i.e., are defined and have non-null values)
- Set default values for variables
- Catch errors that result from variables not being set
- Remove portions of variables' values that match patterns

Syntax of String Operators

The basic idea behind the syntax of string operators is that special characters that denote operations are inserted between the variable's name and the right curly bracket. Any argument that the operator may need is inserted to the operator's right.

The first group of string-handling operators tests for the existence of variables and allows substitutions of default values under certain conditions. These are listed in Table 4-1.*

Table 4-1. Substitution operators

Operator	Substitution
${*varname*:-*word*}	If *varname* exists and isn't null, return its value; otherwise return *word*. **Purpose:** Returning a default value if the variable is undefined. **Example: ${count:-0}** evaluates to 0 if **count** is undefined.
${*varname*:=*word*}	If *varname* exists and isn't null, return its value; otherwise set it to *word* and then return its value. Positional and special parameters cannot be assigned this way. **Purpose:** Setting a variable to a default value if it is undefined. **Example: ${count:=0}** sets **count** to 0 if it is undefined.
${*varname*:?*message*}	If *varname* exists and isn't null, return its value; otherwise print *varname*: followed by *message*, and abort the current command or script (non-interactive shells only). Omitting *message* produces the default message **parameter null or not set**. **Purpose:** Catching errors that result from variables being undefined. **Example: {count:?"undefined!"}** prints "count: undefined!" and exits if **count** is undefined.

* The colon (:) in all but the last of these operators is actually optional. If the colon is omitted, then change "exists and isn't null" to "exists" in each definition, i.e., the operator tests for existence only.

Table 4-1. Substitution operators (continued)

Operator	Substitution
${*varname*:+*word*}	If *varname* exists and isn't null, return *word*; otherwise return null. **Purpose:** Testing for the existence of a variable. **Example: ${count:+1}** returns 1 (which could mean "true") if **count** is defined.
${*varname*:*offset*:*length*}	Performs substring expansion.[a] It returns the substring of $*varname* starting at *offset* and up to *length* characters. The first character in $*varname* is position 0. If *length* is omitted, the substring starts at *offset* and continues to the end of $*varname*. If *offset* is less than 0 then the position is taken from the *end* of $*varname*. If *varname* is @, the *length* is the number of positional parameters starting at parameter offset. **Purpose:** Returning parts of a string (substrings or *slices*). **Example:** If **count** is set to *frogfootman*, **${count:4}** returns *footman*. **${count:4:4}** returns *foot*.

[a] The substring expansion operator is not available in versions of *bash* prior to 2.0.

The first of these operators is ideal for setting defaults for command-line arguments in case the user omits them. We'll use this technique in our first programming task.

Task 4-1

You have a large album collection, and you want to write some software to keep track of it. Assume that you have a file of data on how many albums you have by each artist. Lines in the file look like this:

```
5       Depeche Mode
2       Split Enz
3       Simple Minds
1       Vivaldi, Antonio
```

Write a program that prints the *N* highest lines, i.e., the *N* artists by whom you have the most albums. The default for *N* should be 10. The program should take one argument for the name of the input file and an optional second argument for how many lines to print.

By far the best approach to this type of script is to use built-in UNIX utilities, combining them with I/O redirectors and pipes. This is the classic "building-block" philosophy of UNIX that is another reason for its great popularity with programmers. The building-block technique lets us write a first version of the script that is only one line long:

```
sort -nr $1 | head -${2:-10}
```

Here is how this works: the *sort* program sorts the data in the file whose name is given as the first argument (**$1**). The **-n** option tells *sort* to interpret the first word on each line as a number (instead of as a character string); the **-r** tells it to reverse the comparisons, so as to sort in descending order.

The output of *sort* is piped into the *head* utility, which, when given the argument **-*N***, prints the first *N* lines of its input on the standard output. The expression **-${2:-10}** evaluates to a dash (-) followed by the second argument if it is given, or to -10 if it's not; notice that the variable in this expression is **2**, which is the second positional parameter.

Assume the script we want to write is called *highest*. Then if the user types **highest myfile**, the line that actually runs is:

```
sort -nr myfile | head -10
```

Or if the user types **highest myfile 22**, the line that runs is:

```
sort -nr myfile | head -22
```

Make sure you understand how the **:-** string operator provides a default value.

This is a perfectly good, runnable script—but it has a few problems. First, its one line is a bit cryptic. While this isn't much of a problem for such a tiny script, it's not wise to write long, elaborate scripts in this manner. A few minor changes will make the code more readable.

First, we can add comments to the code; anything between # and the end of a line is a comment. At a minimum, the script should start with a few comment lines that indicate what the script does and what arguments it accepts. Second, we can improve the variable names by assigning the values of the positional parameters to regular variables with mnemonic names. Finally, we can add blank lines to space things out; blank lines, like comments, are ignored. Here is a more readable version:

```
#
#       highest filename [howmany]
#
#       Print howmany highest-numbered lines in file filename.
#       The input file is assumed to have lines that start with
#       numbers.  Default for howmany is 10.
#

filename=$1
howmany=${2:-10}

sort -nr $filename | head -$howmany
```

The square brackets around **howmany** in the comments adhere to the convention in UNIX documentation that square brackets denote optional arguments.

The changes we just made improve the code's readability but not how it runs. What if the user were to invoke the script without any arguments? Remember that positional parameters default to null if they aren't defined. If there are no arguments, then **$1** and **$2** are both null. The variable **howmany** (**$2**) is set up to default to 10, but there is no default for **filename** (**$1**). The result would be that this command runs:

```
sort -nr | head -10
```

As it happens, if *sort* is called without a filename argument, it expects input to come from standard input, e.g., a pipe (|) or a user's terminal. Since it doesn't have the pipe, it will expect the terminal. This means that the script will appear to hang! Although you could always hit CTRL-D or CTRL-C to get out of the script, a naive user might not know this.

Therefore we need to make sure that the user supplies at least one argument. There are a few ways of doing this; one of them involves another string operator. We'll replace the line:

```
filename=$1
```

with:

```
filename=${1:?"filename missing."}
```

This will cause two things to happen if a user invokes the script without any arguments: first the shell will print the somewhat unfortunate message:

```
highest: 1: filename missing.
```

to the standard error output. Second, the script will exit without running the remaining code. With a somewhat "kludgy" modification, we can get a slightly better error message.

Consider this code:

```
filename=$1
filename=${filename:?"missing."}
```

This results in the message:

```
highest: filename: missing.
```

(Make sure you understand why.) Of course, there are ways of printing whatever message is desired; we'll find out how in Chapter 5.

Before we move on, we'll look more closely at the three remaining operators in Table 4-1 and see how we can incorporate them into our task solution. The := operator does roughly the same thing as :-, except that it has the "side effect" of setting the value of the variable to the given word if the variable doesn't exist.

Therefore we would like to use := in our script in place of :-, but we can't; we'd be trying to set the value of a positional parameter, which is not allowed. But if we replaced:

```
howmany=${2:-10}
```

with just:

```
howmany=$2
```

and moved the substitution down to the actual command line (as we did at the start), then we could use the := operator:

```
sort -nr $filename | head -${howmany:=10}
```

The operator **:+** substitutes a value if the given variable exists and isn't null. Here is how we can use it in our example: let's say we want to give the user the option of adding a header line to the script's output. If she types the option **-h**, then the output will be preceded by the line:

```
ALBUMS  ARTIST
```

Assume further that this option ends up in the variable **header**, i.e., **$header** is **-h** if the option is set or null if not. (Later we will see how to do this without disturbing the other positional parameters.)

The following expression yields null if the variable **header** is null, or **ALBUMSARTIST\n** if it is non-null:

```
${header:+"ALBUMSARTIST\n"}
```

This means that we can put the line:

```
echo -e -n ${header:+"ALBUMSARTIST\n"}
```

right before the command line that does the actual work. The **-n** option to **echo** causes it *not* to print a LINEFEED after printing its arguments. Therefore this **echo** statement will print nothing—not even a blank line—if **header** is null; otherwise it will print the header line and a LINEFEED (\n). The **-e** option makes **echo** interpret the \n as a LINEFEED rather than literally.

The final operator, substring expansion, returns sections of a string. We can use it to "pick out" parts of a string that are of interest. Assume that our script is able to assign lines of the sorted list, one at a time, to the variable *album_line*. If we want to print out just the album name and ignore the number of albums, we can use substring expansion:

```
echo ${album_line:8}
```

This prints everything from character position 8, which is the start of each album name, onwards.

If we just want to print the numbers and not the album names, we can do so by supplying the length of the substring:

```
echo ${album_line:0:7}
```

Although this example may seem rather useless, it should give you a feel for how to use substrings. When combined with some of the programming features discussed later in the book, substrings can be extremely useful.

Patterns and Pattern Matching

We'll continue refining our solution to Task 4-1 later in this chapter. The next type of string operator is used to match portions of a variable's string value against *patterns*. Patterns, as we saw in Chapter 1, are strings that can contain wildcard characters (*, **?**, and **[]** for character sets and ranges).

Table 4-2 lists *bash*'s pattern-matching operators.

Table 4-2. Pattern-matching operators

Operator	Meaning
${*variable***#pattern**}	If the pattern matches the beginning of the variable's value, delete the shortest part that matches and return the rest.
${*variable***##pattern**}	If the pattern matches the beginning of the variable's value, delete the longest part that matches and return the rest.
${*variable***%pattern**}	If the pattern matches the end of the variable's value, delete the shortest part that matches and return the rest.
${*variable***%%pattern**}	If the pattern matches the end of the variable's value, delete the longest part that matches and return the rest.
${*variable*/**pattern**/*string*} ${*variable*//**pattern**/*string*}	The longest match to *pattern* in *variable* is replaced by *string*. In the first form, only the first match is replaced. In the second form, all matches are replaced. If the pattern begins with a #, it must match at the start of the variable. If it begins with a %, it must match with the end of the variable. If *string* is null, the matches are deleted. If *variable* is @ or *, the operation is applied to each positional parameter in turn and the expansion is the resultant list.[a]

[a] The pattern-matching and replacement operator is not available in versions of *bash* prior to 2.0.

These can be hard to remember; here's a handy mnemonic device: # matches the front because number signs *precede* numbers; **%** matches the rear because percent signs *follow* numbers.

The classic use for pattern-matching operators is in stripping off components of pathnames, such as directory prefixes and filename suffixes. With that in mind, here is an example that shows how all of the operators work. Assume that the variable **path** has the value */home/cam/book/long.file.name*; then:

```
Expression                  Result
${path##/*/}                     long.file.name
${path#/*/}                cam/book/long.file.name
$path               /home/cam/book/long.file.name
${path%.*}          /home/cam/book/long.file
${path%%.*}         /home/cam/book/long
```

The two patterns used here are **/*/**, which matches anything between two slashes, and .*, which matches a dot followed by anything.

The longest and shortest pattern-matching operators produce the same output unless they are used with the * wildcard operator. As an example, if **filename** had the value **alicece**, then both **${filename%ce}** and **${filename%%ce}** would produce the result **alice**. This is because **ce** is an *exact* match; for a match to occur, the string *ce* must appear on the end **$filename**. Both the short and long matches will then match the last grouping of *ce* and delete it. If, however, we had used the * wildcard, then **${filename%ce*}** would produce **alice** because it matches the shortest occurrence of *ce* followed by anything else. **${filename%%ce*}** would return **ali** because it matches the longest occurrence of *ce* followed by anything else; in this case the first and second *ce*.

The next task will incorporate one of these pattern-matching operators.

Task 4-2

You are writing a graphics file conversion utility for use in creating a web page. You want to be able to take a PCX file and convert it to a JPEG file for use on the web page.[a]

[a] PCX is a popular graphics file format under Microsoft Windows. JPEG (Joint Photographic Expert Group) is a common graphics format on the Internet and is used to a great extent on web pages.

Graphics file conversion utilities are quite common because of the plethora of different graphics formats and file types. They allow you to specify an input file, usually from a range of different formats, and convert it to an output file of a different format. In this case, we want to take a PCX file, which can't be displayed with a web browser, and convert it to a JPEG which can be displayed by nearly all browsers. Part of this process is taking the filename of the PCX file, which ends in *.pcx*, and changing it to one ending in *.jpg* for the output file. In essence, you want to take the original filename and strip off the *.pcx*, then append *.jpg*. A single shell statement will do this:

```
outfile=${filename%.pcx}.jpg
```

The shell takes the filename and looks for **.pcx** on the end of the string. If it is found, **.pcx** is stripped off and the rest of the string is returned. For example, if **filename** had the value **alice.pcx**, the expression **${filename%.pcx}** would return **alice**. The **.jpg** is appended to form the desired **alice.jpg**, which is then stored in the variable **outfile**.

If **filename** had an inappropriate value (without the **.pcx**) such as **alice.xpm**, the above expression would evaluate to **alice.xpm.jpg**: since there was no match, nothing is deleted from the value of **filename**, and **.jpg** is appended anyway. Note, however, that if **filename** contained more than one dot (e.g., if it were **alice.1.pcx—**the expression would still produce the desired value **alice.1.jpg**).

The next task uses the longest pattern-matching operator.

Task 4-3

You are implementing a filter that prepares a text file for printer output. You want to put the file's name—without any directory prefix—on the "banner" page. Assume that, in your script, you have the pathname of the file to be printed stored in the variable **pathname**.

Clearly, the objective is to remove the directory prefix from the pathname. The following line will do it:

```
bannername=${pathname##*/}
```

This solution is similar to the first line in the examples shown before. If **pathname** were just a filename, the pattern ***/** (anything followed by a slash) would not match and the value of the expression would be **pathname** untouched. If **pathname** were something like *book/wonderland*, the prefix *book/* would match the pattern and be deleted, leaving just **wonderland** as the expression's value. The same thing would happen if **pathname** were something like */home/cam/ book/wonderland*: since the ## deletes the longest match, it deletes the entire */home/cam/book/*.

If we used #*/ instead of ##*/, the expression would have the incorrect value *home/ cam/book/wonderland*, because the shortest instance of "anything followed by a slash" at the beginning of the string is just a slash (/).

The construct **${*variable*##*/}** is actually equivalent to the UNIX utility *basename*. *basename* takes a pathname as argument and returns the filename only; it is meant to be used with the shell's command substitution mechanism (see the following explanation). *basename* is less efficient than **${*variable*##*/}** because it runs in its own separate process rather than within the shell. Another utility, *dirname*, does essentially the opposite of *basename*: it returns the directory prefix only. It is equivalent to the *bash* expression **${*variable*%/*}** and is less efficient for the same reason.

The last operator in the table matches patterns and performs substitutions. Task 4-4 is a simple task where it comes in useful.

Task 4-4

The directories in **PATH** can be hard to distinguish when printed out as one line with colon delimiters. You want a simple way to display them, one to a line.

As directory names are separated by colons, the easiest way would be to replace each colon with a LINEFEED:

```
$ echo -e ${PATH//:/'\n'}
/home/cam/bin
/usr/local/bin
/bin
/usr/bin
/usr/X11R6/bin
```

Each occurrence of the colon is replaced by \n. As we saw earlier, the **-e** option allows **echo** to interpret \n as a LINEFEED. In this case we used the second of the two substitution forms. If we'd used the first form, only the first colon would have been replaced with a \n.

Length Operator

There is one remaining operator on variables. It is **${#*varname*}**, which returns the length of the value of the variable as a character string. (In Chapter 6, we will see how to treat this and similar values as actual numbers so they can be used in arithmetic expressions.) For example, if **filename** has the value **alice.c**, then **${#filename}** would have the value **7**.

Extended Pattern Matching

Bash provides a further set of pattern matching operators if the **shopt** option **extglob** is switched on. Each operator takes one or more patterns, normally strings, separated by the vertical bar (|). The extended pattern matching operators are given in Table 4-3.*

Table 4-3. Pattern-matching operators

Operator	Meaning
*(*patternlist*)	Matches zero or more occurrences of the given patterns.
+(*patternlist*)	Matches one or more occurrences of the given patterns.
?(*patternlist*)	Matches zero or one occurrences of the given patterns.
@(*patternlist*)	Matches exactly one of the given patterns.
!(*patternlist*)	Matches anything except one of the given patterns.

Some examples of these include:

- ***(alice|hatter|hare)** would match zero or more occurrences of **alice**, **hatter**, and **hare**. So it would match the null string, **alice**, **alicehatter**, etc.
- **+(alice|hatter|hare)** would do the same except not match the null string.
- **?(alice|hatter|hare)** would only match the null string, **alice**, **hatter**, or **hare**.
- **@(alice|hatter|hare)** would only match **alice**, **hatter**, or **hare**.
- **!(alice|hatter|hare)** matches everything except **alice**, **hatter**, and **hare**.

The values provided can contain shell wildcards too. So, for example, **+([0-9])** matches a number of one or more digits. The patterns can also be nested, so you could remove all files except those beginning with **vt** followed by a number by doing **rm !(vt+([0-9]))**.

Command Substitution

From the discussion so far, we've seen two ways of getting values into variables: by assignment statements and by the user supplying them as command-line arguments

* Be aware that these are not available in early releases of *bash* 2.0.

(positional parameters). There is another way: *command substitution*, which allows you to use the standard output of a command as if it were the value of a variable. You will soon see how powerful this feature is.

The syntax of command substitution is:[*]

```
$(UNIX command)
```

The command inside the parentheses is run, and anything the command writes to standard output is returned as the value of the expression. These constructs can be nested, i.e., the UNIX command can contain command substitutions.

Here are some simple examples:

- The value of **$(pwd)** is the current directory (same as the environment variable **$PWD**).
- The value of **$(ls $HOME)** is the names of all files in your home directory.
- The value of **$(ls $(pwd))** is the names of all files in the current directory.
- The value of **$(< alice)** is the contents of the file *alice* with any trailing newlines removed.[†]
- To find out detailed information about a command if you don't know where its file resides, type **ls -l $(type -path -all** command-name). The **-all** option forces **type** to do a pathname look-up and **-path** causes it to ignore keywords, built-ins, etc.
- If you want to edit (with vi) every chapter of your book on bash that has the phrase "command substitution," assuming that your chapter files all begin with ch, you could type:

  ```
  vi $(grep -l 'command substitution' ch*)
  ```

  The **-l** option to *grep* prints only the names of files that contain matches.

Command substitution, like variable and tilde expansion, is done within double quotes. Therefore, our rule in Chapter 1 and Chapter 3 about using single quotes for strings unless they contain variables will now be extended: "When in doubt, use single quotes, unless the string contains variables or command substitutions, in which case use double quotes."

Command substitution helps us with the solution to the next programming task, which relates to the album database in Task 4-1.

The *cut* utility is a natural for this task. *cut* is a data filter: it extracts columns from tabular data. If you supply the numbers of columns you want to extract from the input, *cut* will print only those columns on the standard output. Columns can be

[*] Bourne and C shell users should note that the command substitution syntax of those shells, **`*UNIX command*`** (with backward quotes, or grave accents), is also supported by *bash* for backward compatibility reasons. However, it is harder to read and less conducive to nesting.

[†] Not available in versions of *bash* prior to 2.02.

Task 4-5

The file used in Task 4-1 is actually a report derived from a bigger table of data about albums. This table consists of several columns, or *fields*, to which a user refers by names like "artist," "title," "year," etc. The columns are separated by vertical bars (|, the same as the UNIX pipe character). To deal with individual columns in the table, field names need to be converted to field numbers.

Suppose there is a shell function called *getfield* that takes the field name as argument and writes the corresponding field (or column) number on the standard output. Use this routine to help extract a column from the data table.

character positions or—relevant in this example—fields that are separated by TAB characters or other delimiters.* Assume that the data table in our task is a file called *albums* and that it looks like this:

```
Depeche Mode|Speak and Spell|Mute Records|1981
Depeche Mode|Some Great Reward|Mute Records|1984
Depeche Mode|101|Mute Records|1989
Depeche Mode|Violator|Mute Records|1990
Depeche Mode|Songs of Faith and Devotion|Mute Records|1993
...
```

Here is how we would use *cut* to extract the fourth (year) column:

```
cut -f4 -d\| albums
```

The **-d** argument is used to specify the character used as field delimiter (TAB is the default). The vertical bar must be backslash-escaped so that the shell doesn't try to interpret it as a pipe.

From this line of code and the *getfield* routine, we can easily derive the solution to the task. Assume that the first argument to *getfield* is the name of the field the user wants to extract. Then the solution is:

```
fieldname=$1
cut -f$(getfield $fieldname) -d\| albums
```

If we called this script with the argument **year**, the output would be:

```
1981
1984
1989
1990
1993
...
```

* Some older BSD-derived systems don't have *cut*, but you can use *awk* instead. Whenever you see a command of the form: `cut -fN -dC filename`, use this instead: `awk -FC '{print $N}' filename`.

Task 4-6 shows another small task that makes use of *cut*.

Task 4-6

Send a mail message to everyone who is currently logged in.

The command *who* tells you who is logged in (as well as which terminal they're on and when they logged in). Its output looks like this:

```
root     tty1         Oct 13 12:05
michael  tty5         Oct 13 12:58
cam      tty23        Oct 13 11:51
kilrath  tty25        Oct 13 11:58
```

The fields are separated by spaces, not TABs. Since we need the first field, we can get away with using a space as the field separator in the *cut* command. (Otherwise we'd have to use the option to *cut* that uses character columns instead of fields.) To provide a space character as an argument on a command line, you can surround it by quotes:

```
$ who | cut -d' ' -f1
```

With the above *who* output, this command's output would look like this:

```
root
michael
cam
kilrath
```

This leads directly to a solution to the task. Just type:

```
$ mail $(who | cut -d' ' -f1)
```

The command **mail root michael cam kilrath** will run and then you can type your message.

Task 4-7 is another task that shows how useful command pipelines can be in command substitution.

Task 4-7

The **ls** command gives you pattern-matching capability with wildcards, but it doesn't allow you to select files by *modification date*. Devise a mechanism that lets you do this.

Here is a function that allows you to list all files that were last modified on the date you give as argument. Once again, we choose a function for speed reasons. No pun is intended by the function's name:

```
function lsd
{
    date=$1
    ls -l | grep -i "^.\{42\}$date" | cut -c55-
}
```

This function depends on the column layout of the **ls -l** command. In particular, it depends on dates starting in column 42 and filenames starting in column 55. If this isn't the case in your version of UNIX, you will need to adjust the column numbers.*

We use the *grep* search utility to match the date given as argument (in the form *Mon DD*, e.g., **Jan 15** or **Oct 6**, the latter having two spaces) to the output of **ls -l**. This gives us a long listing of only those files whose dates match the argument. The **-i** option to *grep* allows you to use all lowercase letters in the month name, while the rather fancy argument means, "Match any line that contains 41 characters followed by the function argument." For example, typing **lsd** 'jan 15' causes *grep* to search for lines that match any 41 characters followed by **jan 15** (or **Jan 15**).†

The output of *grep* is piped through our ubiquitous friend *cut* to retrieve the filenames only. The argument to *cut* tells it to extract characters in column 55 through the end of the line.

With command substitution, you can use this function with *any* command that accepts filename arguments. For example, if you want to print all files in your current directory that were last modified today, and today is January 15th, you could type:

```
$ lp $(lsd 'jan 15')
```

The output of *lsd* is on multiple lines (one for each filename), but LINEFEEDs are legal field separators for the *lp* command, because the environment variable **IFS** (see earlier in this chapter) contains LINEFEED by default.

Advanced Examples: pushd and popd

We will conclude this chapter with a couple of functions that are already built into *bash* but are useful in demonstrating some of the concepts we have covered in this chapter.‡

* For example, **ls -l** on SunOS 4.1.x has dates starting in column 33 and filenames starting in column 46.

† Some older BSD-derived versions of UNIX (without System V extensions) do not support the \{*N*\} option. For this example, use 42 periods in a row instead of **.\{42\}**.

‡ Your copy of *bash* may not have *pushd* and *popd*, since it can be configured without these built-ins.

Task 4-8

The functions *pushd* and *popd* implement a *stack* of directories that enable you to move to another directory temporarily and have the shell remember where you were. Implement them as shell functions.

We will start by implementing a significant subset of their capabilities and finish the implementation in Chapter 6.

Think of a stack as a spring-loaded dish receptacle in a cafeteria. When you place dishes on the receptacle, the spring compresses so that the top stays at roughly the same level. The dish most recently placed on the stack is the first to be taken when someone wants food; thus, the stack is known as a "last-in, first-out" or *LIFO* structure. Putting something onto a stack is known in computer science parlance as *pushing*, and taking something off the top is called *popping*.

A stack is very handy for remembering directories, as we will see; it can "hold your place" up to an arbitrary number of times. The **cd -** form of the **cd** command does this, but only to one level. For example: if you are in *firstdir* and then you change to *seconddir*, you can type **cd -** to go back. But if you start out in *firstdir*, then change to *seconddir*, and then go to *thirddir*, you can use **cd -** only to go back to *seconddir*. If you type **cd -** again, you will be back in *thirddir*, because it is the previous directory.*

If you want the "nested" remember-and-change functionality that will take you back to *firstdir*, you need a stack of directories along with the *pushd* and *popd* commands. Here is how these work:

- The first time **pushd** dir is called, **pushd** pushes the current directory onto the stack, then **cd**s to dir and pushes it onto the stack.
- Subsequent calls to **pushd** dir **cd** to dir and push dir only onto the stack.
- **popd** removes the top directory off the stack, revealing a new top. Then it **cd**s to the new top directory.

For example, consider the series of events in Table 4-4. Assume that you have just logged in, and that you are in your home directory (*/home/you*).

Table 4-4. pushd/popd example

Command	Stack contents	Result directory
pushd lizard	*/home/you/lizard /home/you*	*/home/you/lizard*
pushd /etc	*/etc /home/you/lizard /home/you*	*/etc*
popd	*/home/you/lizard /home/you*	*/home/you/lizard*

* Think of **cd -** as a synonym for **cd $OLDPWD**; see the previous chapter.

Table 4-4. pushd/popd example (continued)

Command	Stack contents	Result directory
popd	*/home/you*	*/home/you*
popd	<empty>	(error)

We will implement a stack as an environment variable containing a list of directories separated by spaces.*

Your directory stack should be initialized to the null string when you log in. To do this, put this in your *.bash_profile*:

```
DIR_STACK=""
export DIR_STACK
```

Do *not* put this in your environment file if you have one. The **export** statement guarantees that DIR_STACK is known to all subprocesses; you want to initialize it only once. If you put this code in an environment file, it will get reinitialized in every subshell, which you probably don't want.

Next, we need to implement *pushd* and *popd* as functions. Here are our initial versions:

```
pushd ()
{
    dirname=$1
    DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
    cd ${dirname:?"missing directory name."}
    echo "$DIR_STACK"
}

popd ()
{
    DIR_STACK=${DIR_STACK#* }
    cd ${DIR_STACK%% *}
    echo "$PWD"
}
```

Notice that there isn't much code! Let's go through the two functions and see how they work, starting with *pushd*. The first line merely saves the first argument in the variable **dirname** for readability reasons.

The second line of the function pushes the new directory onto the stack. The expression **${DIR_STACK:-**$PWD‘ ’} evaluates to **$DIR_STACK** if it is non-null or **$PWD**’’ (the current directory and a space) if it is null. The expression within double quotes, then, consists of the argument given, followed by a single space, followed by DIR_STACK or the current directory and a space. The trailing space on the

* *bash* also maintains a directory stack for the **pushd** and **popd** built-ins, accessible through the environment variable **DIRSTACK**. Unlike our version, however, it is implemented as an *array* (see Chapter 6 for details on arrays).

current directory is required for pattern matching in the *popd* function; each directory in the stack is considered to be of the form "*dirname*".

The double quotes in the assignment ensure that all of this is packaged into a single string for assignment back to DIR_STACK. Thus, this line of code handles the special initial case (when the stack is empty) as well as the more usual case (when it's not empty).

The third line's main purpose is to change to the new directory. We use the **:?** operator to handle the error when the argument is missing: if the argument is given, then the expression **${dirname:?"missing directory name."}** evaluates to **$dirname**, but if it is not given, the shell will print the message **pushd: dirname: missing directory name** and exit from the function.

The last line merely prints the contents of the stack, with the implication that the leftmost directory is both the current directory and at the top of the stack. (This is why we chose spaces to separate directories, rather than the more customary colons as in PATH and MAILPATH.)

The *popd* function makes yet another use of the shell's pattern-matching operators. Its first line uses the # operator, which tries to delete the shortest match of the pattern "* " (anything followed by a space) from the value of DIR_STACK. The result is that the top directory and the space following it are deleted from the stack. This is why we need the space on the end of the first directory pushed onto the stack.

The second line of *popd* uses the pattern-matching operator **%%** to delete the *longest* match to the pattern " *" (a space followed by anything) from DIR_STACK. This extracts the top directory as an argument to **cd**, but it doesn't affect the value of DIR_STACK because there is no assignment. The final line just prints a confirmation message.

This code is deficient in four ways. First, it has no provision for errors. For example:

- What if the user tries to push a directory that doesn't exist or is invalid?
- What if the user tries popd and the stack is empty?

Test your understanding of the code by figuring out how it would respond to these error conditions. The second problem is that if you use *pushd* in a shell script, it will exit everything if no argument is given; **${*varname*:?*message*}** always exits from non-interactive shells. It won't, however, exit an interactive shell from which the function is called. The third deficiency is that it implements only some of the functionality of *bash*'s *pushd* and *popd* commands—albeit the most useful parts. In the next chapter, we will see how to overcome all of these deficiencies.

The fourth problem with the code is that it will not work if, for some reason, a directory name contains a space. The code will treat the space as a separator character. We'll accept this deficiency for now, but you might like to think about how to overcome it in the next few chapters.