CHAPTER 8

The Shell

This chapter introduces the agency that sits between the user and the UNIX system. It is called the *shell.* All the wonderful things you can do with UNIX are possible because this agency understands so much by seeing so little code. It's like an efficient secretary who understands your directives from your gestures, and carries them out by specially devised means you don't need to know. The shell is a command processor; it processes the instructions you issue to the machine.

Conceptually, this is one of the most important chapters of the book, and it implements some of the brilliant ideas of the architects of UNIX. The concepts highlighted here must be understood clearly. They are based on the Bourne shell, named after its founder Steve Bourne. It's the earliest shell that came with the UNIX system. You probably won't be using this shell, but the Bourne shell is the lowest common denominator of them all, and most of its features are available in modern shells as well.

Objectives

- Understand what the shell does to a command. *(8.1)*
- Use *wild-card* characters in matching filenames. *(8.2)*
- Use the `\` to *escape* (remove) the meaning of a special character. *(8.3)*
- Use single and double quotes to protect a group of characters and understand the difference between them. *(8.4)*
- Use the escape sequences used by the **`echo`** command. *(8.5)*
- Understand *streams* and how the shell treats them as files. *(8.6)*
- Redirect *standard output* to a file. *(8.6.1)*
- Redirect *standard input* to originate from a file. *(8.6.2)*
- Redirect *standard error* to a file. *(8.6.3)*
- Understand the significance of the files `/dev/null` and `/dev/tty`. *(8.7)*
- Learn the properties of a *filter* and how the `|` is used to set up a *pipeline* for connecting two or more commands. *(8.8)*
- Use *command substitution* to embed commands in command lines of other commands. *(8.10)*
- Learn the properties of *shell variables*. *(8.11)*
- Learn how commands can be grouped together in a *shell script*. *(8.12)*

Note

You probably don't want to know this right now, but you could also be using any one of these widely used shells—the C shell, Korn shell and bash. Korn and bash are supersets of Bourne, so anything that applies to Bourne also applies to them. However, just a few of the shell's features discussed in this chapter don't apply to the C shell. These differences are noted as and when they are encountered.

To know the shell you are using, invoke the command **echo $SHELL**. The output could show /bin/sh (Bourne shell), /bin/csh (C shell), /bin/ksh (Korn shell) or /bin/bash (bash shell). It does pay to know the shell you are using at this stage.

8.1 The Shell as Command Processor

When you log on to a UNIX machine, you see a prompt. The prompt could be a $, a % or anything; it really doesn't matter for most of this chapter. Even though it may appear that nothing is happening out there, a UNIX command is in fact running. This command is the **shell**. It starts functioning the moment you log in and withers away when you log out.

When you issue a command, the shell is the first agency to acquire the information. It accepts and interprets user requests; these are generally the UNIX commands we key in. The shell examines and rebuilds the command line and then leaves the execution work to the kernel. The kernel handles the hardware on behalf of these commands and all processes in the system. Users can thus afford to remain ignorant of the happenings behind the scenes. This is one of the beauties of UNIX design and philosophy.

The shell is generally *sleeping*. It *wakes* up when input is keyed in at the prompt. (**Sleeping**, **waiting** and **waking** are accepted terms in UNIX parlance.) This input is actually input to the program that represents the shell (**sh** for the Bourne shell). The following activities are typically performed by the shell (Fig. 8.1):

- It issues the prompt ($ or otherwise) and sleeps till you enter a command.
- After a command has been entered, the shell scans the command line for some special characters (*metacharacters*, the focus of attention in this chapter) that have a special meaning for it. Because it permits abbreviated command lines (like the use of * to indicate all files, as in **rm ***), the shell has to make sure the abbreviations are expanded *before* the command can act upon them.
- It then creates a simplified command line and passes it on to the kernel for execution. The shell can't do any work while the command is being executed, and has to wait for its completion.
- After the job is complete, the prompt reappears and the shell returns to its sleeping role to start the next "cycle." You are now free to enter some more commands.

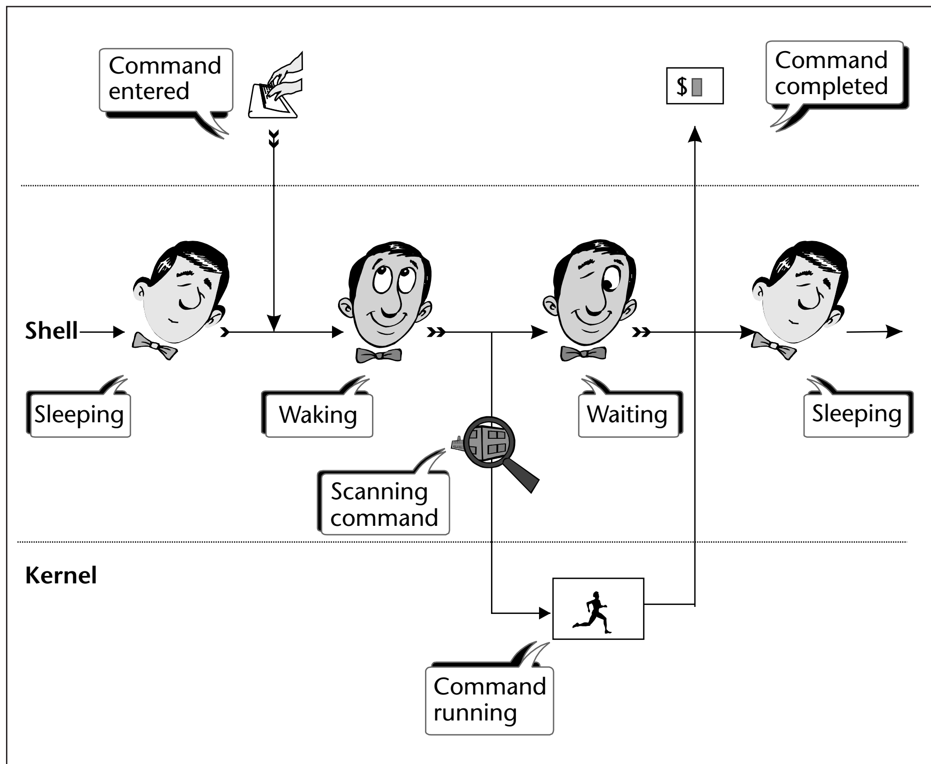
Note that the shell has to interpret these metacharacters because they usually mean nothing to the command. In this chapter, you'll be mainly concerned with the activities that keep the shell preoccupied in its interpretative role.

8.2 Pattern Matching—The Wild Cards

In the last two chapters, you used commands with more than one filename (e.g., **cp chap01 chap02 chap03 progs**) as arguments. Often, you'll need to enter a number of similar filenames in the command line:

```
ls -l chap chap01 chap02 chap03 chap04 chapx chapy chapz
```

FIGURE 8.1 *The Shell's Interpretive Cycle*



Since the filenames here have a common string `chap`, the lengthy command line using this string repeatedly looks rather wasteful. Why can't we have a single pattern comprising the string `chap`, along with one or two special characters? Fortunately, the shell does offer such a solution.

The shell recognizes some characters as special. You can use them to devise a generalized pattern or model that can often match a group of similar filenames. In that case, you can use this pattern as an argument to a command rather than supply a long list of filenames which the pattern represents. The shell itself performs this expansion on your behalf and supplies the expanded list to the command.

8.2.1 The * and ?

Now, let's get into the specifics. In Chapter 6, you used the command **`rm *`** *(6.12)* to delete all files in the current directory. The *, known as a *metacharacter,* is one of the characters of the shell's special set. This character matches any number of characters (including none).

When the * is appended to the string `chap`, the pattern `chap*` matches filenames beginning with the string `chap`—including the file `chap`. It thus matches all the files specified in the previous command line. You can now use this pattern as an argument to **`ls`**:

```
$ ls -x chap*
chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy
chapz
```

When the shell encounters this command line, it immediately identifies the * as a metacharacter. It then creates a list of files from the current directory that match this pattern. It reconstructs the command line as below, and passes it on to the kernel for execution:

```
ls -x chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz
```

What happens when you use **echo** with the * as argument?

```
$ echo *
array.pl back.sh calendar cent2fah.pl chap chap01 chap02 chap03 chap04 chap15 ch
ap16 chap17 chapx chapy chapz count.pl date_array.pl dept.lst desig.lst n2words.
pl name.pl name2.pl odfile operator.pl profile.sam rdbnew.lst rep1.pl
```

You simply see a list of files! The shell uses the * to match files in the current directory. All files match, so you see all of them in the output.

Note

Windows users may be surprised to know that the * may occur anywhere in a filename, and not merely at the end. Thus, `*chap*` matches all the following filenames—`chap newchap chap03 chap03.txt`.

The next metacharacter is the `?`. This matches a single character. When used with the same string `chap` (as `chap?`), the shell matches all five-character filenames beginning with `chap`. Place another `?` at the end of this string, and you have the pattern `chap??`. Use both these expressions separately, and the meaning of the `?` becomes obvious:

```
$ ls -x chap?
chapx chapy chapz
$ ls -x chap??
chap01 chap02 chap03 chap04 chap15 chap16 chap17
```

These metacharacters relating to filenames are also known as **wild cards** (something like the joker that can match any card). The complete list of the shell's wild cards is shown with examples in Table 8.1. We'll now take up the significance of the other wild cards.

Note

The wild-card characters the shell uses to match patterns bear some resemblance to the ones used by **vi** and **emacs** in their regular expressions. But make no mistake, the similarities are only superficial. Regular expressions are understood and interpreted by the command (like **vi** and **emacs**) and have nothing to do with the shell.

8.2.2 The Character Class

Note that the patterns framed in the preceding examples are not very restrictive. It's not easy to list only the files `chapy` and `chapz` with a compact expression. Nor is it easy to pick out only the first four chapters from the numbered list. You need the **character class** for this matching work.

TABLE 8.1 *The Shell's Wild Cards and Application*

Wild Card	Significance
*	Matches any number of characters including none
?	Matches a single character
[*ijk*]	Matches a single character—either an *i, j* or *k*
[!*ijk*]	Matches a single character that is *not* an *i, j* or *k*
[*x*-*z*]	Matches a single character that is within the ASCII range of the characters *x* and *z*
[!*x*-*z*]	Matches a single character that is *not* within the ASCII range of the characters *x* and *z*

Examples

Command	Significance
`ls *.lst`	Lists all files with extension `.lst`
`mv * ../bin`	Moves all files to `bin` subdirectory of parent directory
`compress .?*.?*`	Compresses all files beginning with a dot, followed by one or more characters, then a second dot followed by one or more characters
`>cp foo foo*`	Copies file `foo` to file `foo*` (* loses meaning here)
`cp ?????? progs`	Copies to progs directory all six-character filenames
`cmp rep[12]`	Compares files `rep1` and `rep2`
`rm note[0-1][0-9]`	Removes files `note00`, `note01` . . . . through `note19`
`lp *.[!o]`	Prints all files except C object files (with `.o` extension)
`cp ?*.*[!1238] ..`	Copies to the parent directory files having extensions with at least one character before the dot, but not having 1, 2, 3 or 8 as the last character

The character class uses two more metacharacters represented by a pair of brackets [ ]. You can have multiple characters inside this enclosure, but matching takes place for a single character in the class. For example, a single character expression that can take one of the values 1, 2 or 4, can be represented by the expression

```
[124]                                                    Either 1, 2 or 4
```

This can be combined with any string or another wild-card expression, so selecting the files `chap01`, `chap02` and `chap04` now becomes a simple matter:

```
$ ls -x chap0[124]
chap01  chap02  chap04
```

You can specify ranges inside the class with a `-` (hyphen); `[a-h]` is a character class using a range. This is normally done with numerals and alphabets because these are the characters mostly used in filenames. So, to select the first four numbered chapters, you have to use the range `[1-4]`:

```
$ ls -x chap0[1-4]
chap01  chap02  chap03  chap04
```

A valid range specification requires that the character on the left have a lower ASCII value than the one on the right. Using this property, the files `chapx`, `chapy` and `chapz` can also be listed in a similar manner:

```
$ ls -x chap[x-z]
chapx  chapy  chapz
```

Note

The expression `[a-zA-Z]*` matches all filenames beginning with an alphabet, irrespective of case. You can match a word character by including numerals and the underscore character as well—`[a-zA-Z0-9_]`.

8.2.3 Negating the Character Class

The `!` (bang) is the last character in the set of wild cards. Placing it at the *beginning* of this class reverses the matching criteria, i.e., it matches all other characters except the ones in the class. To reverse the earlier example made in the note, the pattern

```
[!a-zA-Z]*
```

matches all files where the first character is not alphabetic.

The character class used with a `!` represents the only means of negating a single character. To match all files except those having the `.Z` extension (those compressed with the **`compress`** command), you can use this pattern:

```
*.[!Z]                                    All files not ending with .Z
```

When organizing information in groups of files, you should choose the filenames with care so that one, or at most two, metacharacter patterns can match all of them. If you don't do that, be prepared to specify them separately every time you use a command that accesses all of them!

8.2.4 When Wild Cards Lose Their Meaning

Now that you have seen all the wild cards, you should also know that some of these characters have different meanings depending on where they are placed in the pattern. It's important that you know them because you may sometimes find it difficult to match some filenames with a wild card pattern.

The metacharacters `*` and `?` lose their meaning when used inside the class, and are matched literally. Similarly, `-` and `!` also lose their significance when placed outside the class. Additionally, `!` loses its meaning when placed anywhere but at the beginning of the class. The `-` also loses its meaning if it is not bounded properly on either side by a single character.

Note

`[!!]` matches a single character filename that is not a `!`. This doesn't work in the C shell and bash, which use the `!` for a different purpose.

8.2.5 Matching the Dot

There are further restrictions. The * doesn't match all files *beginning* with a **.** (dot) or the / of a pathname. If you want to list all the hidden files in your directory having at least three characters after the dot, then the dot must be matched explicitly:

```
$ ls -x .???*
.exrc .news_time .profile
```

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly. For example, the expression `emp*lst` matches a dot embedded in the filename:

```
$ ls -x emp*lst
emp.lst emp1.lst emp22lst emp2.lst empn.lst
```

Note

The * doesn't match all files beginning with a dot. Such files must be matched explicitly. One *, however, can match any number of embedded dots. For instance, the pattern **fw*gz** matches the filename **fwtk2.1.tar.gz**.

8.2.6 When Using `rm` with *

While these metacharacters help speed up your interaction with the system, there are great risks involved if you are not alert enough. A word of caution at this stage should be appropriate. When using shell wild cards, especially the *, you could be totally at sea if, instead of typing

```
rm chap*
```

which removes all the chapters, you inadvertently introduce a space between `chap` and *:

```
$ rm chap *                                    Very dangerous!
rm: chap: No such file or directory
```

The error message here masks a disaster that has just occurred; the **rm** command has removed all files in this directory! A singular * used with **rm** can be extremely dangerous as the shell treats it as a separate argument. In such situations, you should pause and check the command line before you finally press *[Enter]*.

What if the shell fails to match a single file with the expression `chap*`? There's a surprise element here; the shell also looks for a file named `chap*`. You should avoid using metacharacters when choosing filenames, but if you have to handle one, then you have to turn off the meaning of the * so that the shell treats it literally. This deactivation feature is taken up in the next section.

Wild cards represent a feature of the shell and not of the command using them. The shell has to do the wild-card expansion because `chap*` means nothing to **rm**—nor to any command that uses a filename as argument. The design of the UNIX system prevents the execution of a command till the shell has expanded all wild-card expressions.

Note

It's not wholly true to suggest that wild cards mean nothing to a command, but only to the shell. The **find** command *(7.15)* accepts wild cards (probably the only UNIX command having this feature) as parameters to the -name keyword:

```
find / -name "*.[hH][tT][mM][lL]" -print          All .html and .HTML files
find . -name "note??" -print                       Two characters after note
```

Here, we are using the same wild-card characters, but this time they are a feature of the find command, and not of the shell. By providing quotes around the pattern, we ensured that the shell can't even interpret this pattern. You'll learn about this insulating feature shortly.

Tip

To eliminate the danger of accidental deletion of your files, it makes sense to customize the **rm** command so that it always invokes the **rm -i** command. You can then make a decision on each file individually. This requires the use of an *alias (17.4)*, which is supported by the other shells. The alias definitions can be placed in a startup file *(17.9)*, which the shell reads every time a user logs in.

8.3 Escaping—The Backslash (\)

It's a generally accepted principle that filenames shouldn't contain the shell metacharacters. What happens if they do? The answer is tricky, and can cause a great deal of havoc before you realize it fully. Imagine a file named chap* created with the > symbol:

```
$ echo > chap*                                      Creates an empty file
$ _
```

The silent return of the prompt suggests that the file has been created. A suitable wild-card pattern used with **ls** confirms this:

```
$ ls -x chap*
chap   chap*  chap01  chap02  chap03  chap04  chap15  chap16  chap17  chapx
chapy  chapz
```

There's indeed a file with the name chap* in the current directory! The wild-card pattern matched this file along with the others. This file can be a great nuisance and should be removed immediately. But that won't be easy. You can't use **rm chap*** because that would remove all files in this list, and not this one only.

How do you remove this file then, without deleting the other files? For this to be possible, the shell has to treat the asterisk literally instead of interpreting it as a metacharacter. The answer lies in the \ (backslash)—yet another metacharacter, but one that removes the meaning of any metacharacter placed after it. Use the \ before the * and it solves the problem:

```
$ ls -x chap\*                                      Literally matches chap*
chap*
$ rm chap \*
$ ls -x chap\*
chap* not found
```

The expression chap* literally matches the string chap*. This is a necessary feature provided by the shell, and you'll see how this concept can be extended to other areas also. The use of the \ in removing the magic from any special character is called **escaping** or **despecializing**.

If you have the files chap01, chap02 and chap03 in your current directory, and then create a file chap0[1-3] by using

```
echo > chap0[1-3]
```

then you should escape the two rectangular brackets when accessing the file:

```
$ ls -x chap0\[1-3\]
chap0[1-3]
$ rm chap0\[1-3\]                                Deletes chap0[1-3]— one file
$ ls -x chap0\[1-3\]
chap0[1-3] not found                             File removed
```

Sometimes, you would need to escape the \ character itself. Since the shell treats this as a special character, you need another \ to escape it:

```
$ echo \\
\
$ echo "The newline character is \\n"
The newline character is \n
```

Apart from the wild cards, there are other characters that the shell considers special. Many of them will often need escaping. Here are five of them:

```
$ echo \|\<\>\'\"
|<>'"
```

The shell uses these characters for its interpretive work. The |, < and > are required for handling command input and output. The ' and " also protect special characters, but a group of them. We'll take a detailed look at these characters in this chapter.

8.3.1 Escaping the *[Enter]* Key

Apart from these wild cards, there are other characters that are special to the shell—the newline character, for example. When you enter a long chain of commands or a command with numerous arguments, you can split the command line by hitting *[Enter]*, but only after the \ escapes this key:

```
$  find /usr/local/bin /usr/bin -name "*.pl" -mtime +7 -size -1024 \[Enter]
>  -size +2048 -atime +25 -print                 Note the >
```

This is the **find** command at work—a command often used with several arguments. Escaping is the best way of imparting readability to these lengthy command lines. The \ here escapes the meaning of the newline character generated by *[Enter]*. It also pro-

duces the second prompt (which could be a > or a ?), which indicates that the command line is incomplete.

Escaping is an unsatisfactory solution when you need to despecialize the meaning of a group of characters instead of a single one. Quoting is a better alternative.

Note

The second prompt could be a > or a ?, but in either case it implies that the command is not complete. The C shell uses the ?, but other shells use the >.

8.4 Quoting

There's another way to turn off the meaning of a special character. When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off:

```
$ echo '*?[8-9]'                    Can use double quotes also
*?[8-9]
```

The argument above is said to be **quoted**. Double quotes would also have served the purpose, but in some cases, use of double quotes does permit interpretation of some of the special characters (especially the $ and `). This will become clear as more features of the shell are exposed. For a beginner, single quotes are the safest as they protect all special characters (except the quotes themselves!).

8.4.1 Quoting Preserves Spaces

The space is another character that has a special meaning to the shell. When the shell finds contiguous spaces and tabs in the command line, it compresses them to a single space. When you issue the following **echo** command, you find all spaces compressed:

```
$ echo The shell       compresses    multiple    spaces
The shell compresses multiple spaces
```

The above arguments to **echo** could have been preserved by escaping the space character wherever it occurs at least twice:

```
$ echo The shell \ \ \ compresses \ \ multiple \ \ spaces
The shell       compresses  multiple  spaces
```

When you have a large number of characters which you need to protect from the shell, quoting is preferable to escaping:

```
$ echo "The shell     compresses multiple    spaces"
The shell       compresses  multiple  spaces
```

We used double quotes this time, and it worked just as well. Quotes also protect the \:

```
$ echo '\'
\
```

echo is an unusual command. So far, we used the \ to keep the shell out of the picture. The same character can be used to make **echo** behave differently—while the shell continues to stay out. This feature follows next.

8.5 Escaping and Quoting in echo

Apart from the shell, there are some commands that use the \ as part of their syntax. Rather than remove the special meaning, the \ is used to *emphasize* a character so that the command (and not the shell) treats it as special. Consider, for instance, the following **echo** command:

```
$ echo 'Enter Your Name : \c'
Enter Your Name : $ _
```

Observe that the prompt has been returned, not in the next line, but at the end of the echoed string. The shell can't interpret the \ this time because of the quotes. It's **echo** that interprets it and treats the character c as special. \c used here represents an **escape sequence**, which positions the cursor immediately after the argument instead of the next line.

echo also accepts other escape sequences that manipulate the cursor motion in a number of ways:

\t—A tab
\f—A formfeed (page skip)
\n—A newline

This is how they are used:

```
$ echo '\tThis message is broken here\n\ninto three lines'
        This message is broken here                    Tab in effect
                                     A blank line splits the message
into three lines
```

echo also accepts ASCII octal values as arguments. For instance, *[Ctrl-g]* results in the sounding of a beep. This key has the octal value 007. You can use this value as an argument to the command, but only after preceding it with a \:

```
$ echo '\007'                          Double quotes will also do
... beep heard ...
```

This is the first time we see ASCII octal values used by a UNIX command. (Very few commands use them.) Some people use **echo** to display the box-drawing characters on their terminal using their ASCII values.

BASH Shell

The escape sequences described with **echo** won't work in this form with the bash shell used in Linux. For using them, **echo** must be used with the -e option also:

```
echo -e "Enter your name:\c"
```

We'll be using these escape sequences extensively in this text, so if you are a Linux user, you must commit this option to memory. We'll also be designing a script *(19.13)* that checks the shell that is used and inserts this option automatically!

8.6 Redirection

Many of the commands that we used sent their output to the terminal. You've seen the **cat** *(6.14.1)* and **bc** *(3.12)* commands also taking input from the keyboard. Were these commands designed that way to accept only fixed sources and destinations? No, far from it. They are actually designed to use a **character stream** without knowing its source and destination. A stream is just a sequence of bytes that many commands see as input and output.

UNIX treats these streams as files, and a group of UNIX commands reads from and writes to these files. A command is usually not designed to send output to the terminal—but to this file. Likewise, it is not designed to accept input from the keyboard either—but only from a standard file which it sees as a stream. There's a third stream for all error messages thrown out by a program. This stream is the third file.

It's here that the shell comes in. The shell sets up these three standard files (for input, output and error) and attaches them to a user's terminal at the time of logging in. Any program that uses streams will find them open and available. The shell also closes these files when the user logs out.

The standard file for input is known as *standard input* and that for output is known as *standard output*. The error stream is known as *standard error*. By themselves, these standard files are not associated with any physical device, but the shell has set some physical devices as defaults for them:

- Standard input—the default source is the keyboard.
- Standard output—the default destination is the terminal.
- Standard error—the default destination is the terminal.

It's by design that both standard output and standard error share the same default device—the terminal. In the ensuing topics, you'll see how the shell reassigns (replaces) any of these files by a physical file in the disk the moment it sees some special characters in the command line. This means that instead of input coming from the keyboard and output and error going to the terminal, they can be **redirected** to come from or go to any disk file or some other device.

8.6.1 Standard Output

Commands like **cat** and **who** send their output as a character stream. This stream is called the **standard output** of the command—by default, it appears on the terminal. Using the symbols > and >>, you can redirect the output to a disk file. We'll now do that with the **who** command:

```
$ who > newfile
$ _                                                  The prompt just returns
```

The shell looks at the >, understands that standard output has to be redirected, opens the file newfile, writes the stream into it and then closes the file. And all this happens without **who** knowing absolutely anything about it! newfile now contains a list of users (the output of **who**). This is the way we save command output in files.

If the output file doesn't exist, the shell creates it *before* executing the command. If it exists, the shell overwrites it; so use this operator with caution. Alternatively, you can append to a file using the >> (the right chevron used twice) symbols:

```
who >> newfile                                Doesn't disturb existing contents
```

Redirection also becomes a useful feature when concatenating the standard output of a number of files. Using wild cards you can set up an abbreviated command line:

```
cat chap?? > textbook
```

You can also combine two or more commands and redirect their aggregate output to a file. A pair of parentheses groups the commands, and a single > symbol can be used to redirect both of them:

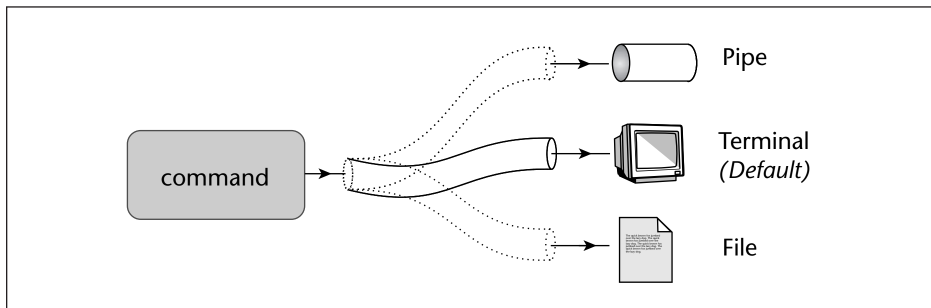
```
( ls -l ; who ) > lsfile
```

The previous chapters couldn't prove convincingly that UNIX makes very little distinction between various types of files. Redirection often doesn't care about file type. It can work with a device name to echo a message on someone's terminal. The following command redirects a message to the terminal /dev/tty02, and a user working on this terminal could see it (provided the terminal is set up accordingly).

```
echo This message is for the terminal tty02 >/dev/tty02
```

The terminal is the first (and default) destination of standard output. The disk file is the second. There's a third destination—as input to another program, which we'll take up when discussing pipelines. The handling of the standard output stream using these three destinations is shown in Fig. 8.2.

FIGURE 8.2 *The Three Destinations of Standard Output*



Note

When the output of a command is redirected to a file, the output file is created by the shell *before* the command is executed. Any idea what **cat foo > foo** does?

8.6.2 Standard Input

Some commands are designed to take their input also as a stream. This stream represents the **standard input** to a command. You used the **wc** command *(1.10.5)* for counting lines, words and characters in a file. But the same command expects input from the keyboard when the filename is omitted; you have to key it in:

```
$ wc                                   No filename!
      2 ^ 32                           Spaces provided deliberately
     25 * 50
30*25 + 15^2
[Ctrl-d]
      3       9      39                No filename in output
```

You used **cat** *(6.14.1)* in this way too. Enter the three lines of text (a group of mathematical expressions), signify the end of input with *[Ctrl-d]*, and then press *[Enter]*. **wc** immediately counts 3 lines, 9 words and 39 characters in its standard input.

This input can be similarly redirected to originate from a file (the second source). First fill up the file `calc.lst` with the three expressions (using **cat > calc.lst**). Now, when the metacharacter < (left chevron) is used in this way, the shell redirects **wc**'s standard input to come from this file:

```
$ wc < calc.lst
      3       9      39
```

This too is standard input, but in its second form. Note once more that **wc** didn't open the file. It can do so, but only when it uses a filename as an argument:

```
$ wc calc.lst
      3       9      39 calc.lst
```

Note that **wc** this time shows the filename; it very well can because it opens the file itself.

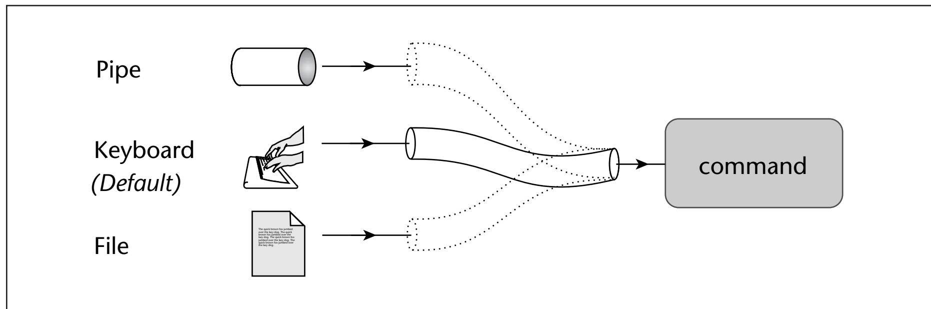
You may have already framed your next question. Why bother to redirect the standard input from a file if the command can read the file itself as above? The answer is that there are times when you need to keep the command ignorant of the source of its input. This aspect, representing one of the most deep-seated features of the system, will gradually expose itself as you progress through these chapters.

To sum up, the standard input stream also has three sources:

- The keyboard, the default source.
- A file using redirection with <.
- Another program using a pipeline (to be taken up later).

The handling of the standard input stream from these three sources is shown in Fig. 8.3.

FIGURE 8.3 *The Three Sources of Standard Input*



Note

When the standard input is redirected to come from a file (with <), it's the shell that opens the file. The command here is totally ignorant of the shell's activities. However, when the filename is supplied as an argument to a command, it's the command which opens the file and not the shell.

Making Calculations in a Batch You have used the **bc** command *(3.12)* as a calculator. This command still has a few surprises in store for us. Take, for instance, the file `calc.lst` that contains some arithmetic expressions. You can redirect `bc`'s standard input to come from this file:

```
$ bc < calc.lst > result.lst
$ cat result.lst
4294967296                    This is 2^32
1250                          This is 25*50
975                           This is 30*25 + 15^2
```

Look what's happened here. **bc** took each line from `calc.lst` and evaluated it. You don't need to perform your calculations "on-line" anymore. You can place them in a file and run the whole job as a batch. You can also save the output in a separate file (using >). It would be better still if we could have each expression in `calc.lst` beside the computed result. We'll do that too, but only after we learn how to use the **paste** command.

Input Both from File and Standard Input When a command takes input from multiple sources—say a file and standard input, the - symbol must be used to indicate the sequence of taking the input. The meaning of the following sequences should be quite obvious:

```
cat - foo          First from standard input and then from foo
cat foo - bar      First from foo, then standard input, and then bar
```

There's a fourth form of standard input which we have not considered here. It's the *here document* that has application in shell programming and hence discussed in Chapter 19.

8.6.3 Standard Error

When you enter an incorrect command or try to open a nonexistent file, certain diagnostic messages show up on the screen. This is the **standard error** stream. Like standard output, it too is destined for the terminal. Note that they are in fact two separate streams, and the shell possesses a mechanism for capturing them individually. Trying to "cat" a nonexistent file produces the third stream:

```
$ cat bar
cat: cannot open bar: No such file or directory
```

The standard error stream can also be reassigned to a file. Using the symbol for standard output obviously won't do:

```
$ cat bar > errorfile
cat: cannot open bar: No such file or directory
```

You tried to "cat" a file that doesn't exist, but the error message still shows up on the terminal. Before we proceed any further, you should know that each of these three standard files has a number, called a **file descriptor**, which is used for identification:

0—Standard input *`<` is same as `0<`*
1—Standard output *`>` is same as `1>`*
2—Standard error *Must be `2>` only*

These descriptors are implicitly prefixed to the redirection symbols. For instance, `>` and `1>` mean the same thing to the shell, while `<` and `0<` also are identical. You normally don't need to use the numbers `0` and `1` to prefix the redirect symbols because they are the default values. However, we need to use the descriptor `2>` for the standard error:

```
$ cat bar 2>errorfile
$ cat errorfile
cat: cannot open bar: No such file or directory
```

This works. You can also append diagnostic output in a manner similar to the one in which you append standard output:

```
cat bar 2>> errorfile
```

You can now save error messages in a separate file. This enables you to run long programs and save error output to be viewed at the end of the day.

C Shell

The standard error is handled differently by the C shell, so the examples of this section won't work with it. In fact, the C shell merges the standard error with the standard output; it has no separate symbol for handling standard error only.

8.6.4 Combining Streams

When a command is used without a filename or with a `-`, it signifies that the input is from the standard input. You can redirect the output too. These two forms are equivalent:

```
cat > foo                                          - is not necessary here
cat - > foo                                        Both input and output redirected
```

You used the first command *(6.14.1)* to create a file by redirecting both input and output. You can also combine the < and > operators; there's no restriction imposed on their sequence either:

```
wc < infile > newfile                              First input, then output
wc>newfile<infile                                  First output, then input
> newfile < infile wc                              As above, but command at end
```

The <, > and the >> operators are indifferent to the presence of spaces around them. In all these cases, the shell keeps the command ignorant of both source and destination. The last example illustrates a significant departure from a statement made previously *(2.4)* that the first word in the command line is the command. In the last example, **wc** is the last word in the command line.

The standard output and error symbols can also be used in the same command line:

```
cat newfile nofile 2> errorfile > outfile
```

Sometimes, you'll need to direct both the standard output and standard error streams to the same file. You'll have to use some more special symbols, and you'll learn about them in Chapter 19.

8.6.5 A New Categorization of Commands

Do all commands use the features of standard input and standard output? No, certainly not. From this viewpoint, the UNIX commands can be grouped into four categories:

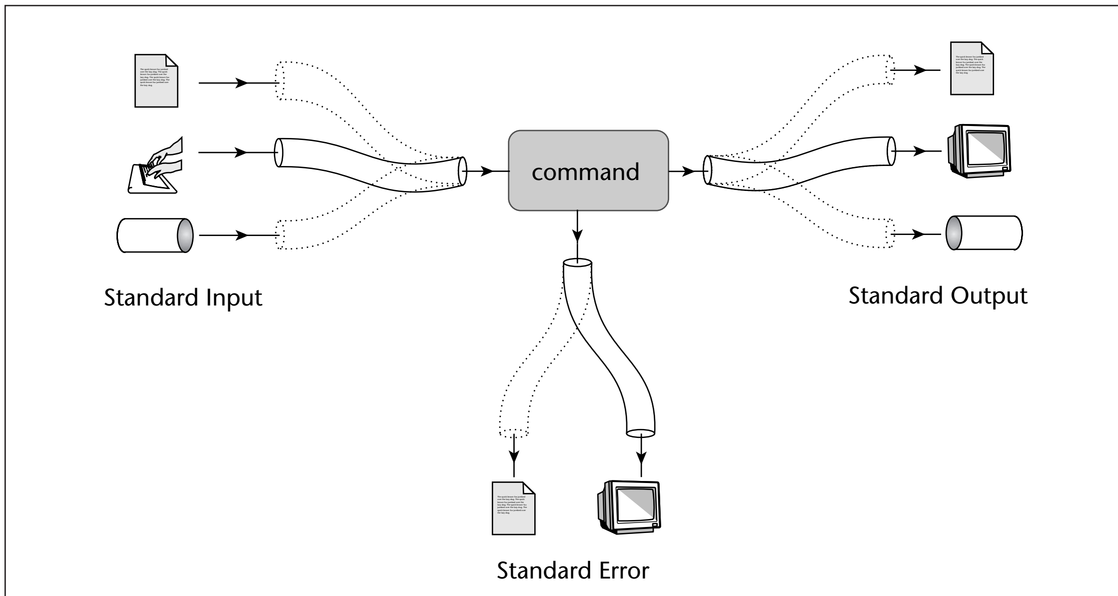
Commands	Standard Input	Standard Output
mkdir, rmdir, cp, rm	No	No
ls, pwd, who	No	Yes
lp, lpr	Yes	No
cat, bc, wc	Yes	Yes

The indifference of a command to the source of its input and destination of its output is one of the most profound features of the UNIX system. It raises the possibility of commands "talking" to one another, so that the output of one command can be used as the input to another. We'll set up pipelines later that permit this communication. The handling of the three streams is shown in Fig. 8.4.

8.7 /dev/null and /dev/tty: Two Special Files

Quite often, you may want to test whether a program runs successfully without seeing the output or the error messages on the screen. You may not want to save this output in files either. You have a special file that simply accepts any stream without growing in size—the file /dev/null:

FIGURE 8.4 *The Three Standard Files*




```
$ cal 1995 >/dev/null
$ cat /dev/null                         Size is always zero
$ _
```

The device file `/dev/null` simply incinerates all output directed towards it. Its size always remains zero. This facility is useful in redirecting error messages away from the terminal so that they don't appear on the screen. The following sequence attempts to "cat" a nonexistent file without cluttering the display:

```
cat chap100 2>/dev/null                 Redirects the standard error
```

`/dev/null` is actually a pseudo-device because, unlike all other device files, it's not associated with any physical device.

The second special file in the UNIX system is the one indicating one's terminal—`/dev/tty`. Consider, for instance, that romeo is working on terminal `/dev/tty01` and juliet on `/dev/tty02`. However, both romeo and juliet can refer to their own terminals with a single device file—`/dev/tty`. Thus, if romeo issues the command

```
who >/dev/tty
```

the list of current users is sent to the terminal he is currently using—`/dev/tty01`. Similarly, juliet can use an identical command to see the output on her terminal `/dev/tty02`. Like `/dev/null`, `/dev/tty` is another special file that can be accessed independently by several users without conflict.

You may ask why one should need to specifically redirect any output to one's own terminal since the default output goes to the terminal anyway. Sometimes, you do need to specify that explicitly. Apart from its use in redirection, this file can also be used as an argument to some UNIX commands. Section 8.9 makes use of this feature, while some situations are presented in Chapter 19 (featuring shell programming).

Tip

If you use `find` from an ordinary nonprivileged account to start its search from root, the command will generate a lot of error messages on being unable to "cd" to a directory. Since you might miss the selected file in an error-dominated list, the standard error of **`find`** should be directed to `/dev/null`—like `find / -name typescript -print 2>/dev/null`.

8.8 Pipes

To understand pipes, we'll set ourselves the task of counting the number of users currently logged in. We'll first attempt the task using the knowledge we possess already. **`who`** produces a list of users—one user per line, and we'll save this output in a file:

```
$ who > user.lst
$ cat user.lst
romeo      tty01       May 18 09:32
juliet     tty02       May 18 11:18
andrew     tty03       May 18 13:21
```

If we now redirect the standard input of the **`wc -l`** command *(1.10.5)* to come from `user.lst`, we would have effectively counted the number of users:

```
$ wc -l < user.lst
      3                                              The number of users
```

This method of using two commands in sequence has certain obvious disadvantages:

- The process is slow. The second command can't act unless the first has completed its job.
- You require an intermediate file that has to be removed after the **`wc`** command has completed its run.
- When handling large files, temporary files can build up easily and eat up disk space in no time.

Here, **`who`**'s standard output was redirected, and so was **`wc`**'s standard input. You may ask: Can't the shell connect these streams together so that one command takes input from the other? Yes, the shell can, using a special operator as the connector of two commands—the | (pipe). You can make **`who`** and **`wc`** work in tandem so that one takes input from the other:

```
$ who | wc -l
      3
```

Here, **`who`** is said to be *piped* to **`wc`**. No intermediate files are created when they are used. When a sequence of commands is combined together in this way, a **pipeline** is

said to be formed. The name is appropriate as the connection it establishes between programs resembles a plumbing joint. It's the shell that sets up this interconnection, and the commands have no knowledge of it.

The pipe is the third source and destination of standard input and standard output, respectively. You can now use one to count the number of files in the current directory:

```
$ ls | wc -l
    15
```

Note that no separate command was designed to tell you that, though the designers could easily have provided another option to **ls** to perform this operation. And because **wc** uses standard output, you can redirect this output to a file:

```
ls | wc -l > fkount
```

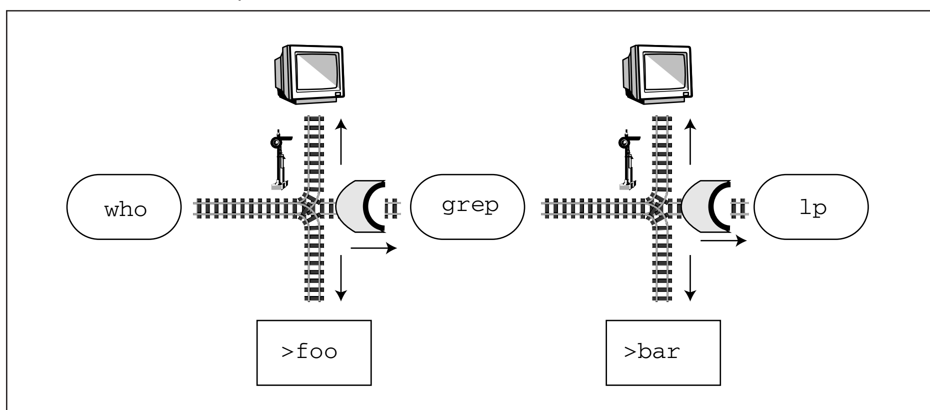
There's no restriction on the number of commands you can use in a pipeline. But you must know the behavioral properties of these commands to place them there. Consider this generalized command line:

command1 | *command2* | *command3* | *command4*

It should be pretty obvious that *command2* and *command3* must support both standard input and standard output. *command1* requires to use standard output only, while *command4* must be able to read from standard input. If you can ensure that, then you can have a chain of these tools connected together as shown in Fig. 8.5. The commands *command2* and *command3* who support both streams are called **filters**. Filters are the central tools of the tool kit, and are discussed later in four entire chapters.

Printing the man Pages The online man pages of a command often show the keywords in boldface. These pages contain a number of control characters which have to be removed before you can print them. The **col -b** command can remove these characters from its input, which means that the **man** output has to be piped to **col -b**:

FIGURE 8.5 *A Pipeline of Three Commands*



```
man grep | col -b > grep.txt
```

This sequence sends clear text to a text file, but we can pipe it again to print the page. The **lp** command *(6.16)* prints a file, but also accepts standard input:

```
man grep | col -b | lp
```

8.8.1 When a Command Needs to Be Ignorant of Its Source

Now where does all this discussion on redirection and piping lead us to? Let's consider the **grep** command which uses a filename as argument:

```
$ grep "print" foo1                                  grep opens the file foo1
print "Content-type: text/html\n\n";
print "</html>\n";
```

grep *(15.2)* locates lines containing a specified pattern in its input. Here it displays lines containing the string `print`. Since **grep** is also a filter, it should be able take input from the standard input as well:

```
grep "print" < foo1                                  Shell opens the file foo1
```

This also produces the same output, so what difference does it really make? This question was posed before *(8.6.2)*, but we'll have to answer it this time. To know why we sometimes need **grep** to handle a stream rather than a file, consider that **grep** also accepts multiple filenames:

```
$ grep "print" foo1 foo2 foo3
foo1:print "Content-type: text/html\n\n";
foo1:print "</html>\n";
foo2:find / -mtime +7 -print | perl -ne 'chop ; unlink ;'
foo3:$slno++ ; print ($slno . " " . $_ . "\n") ;
foo3:printf "File $file was last modified %0.3f days back \n", $m_age ;
```

grep prints the filenames this time; it can since it opens the files and knows their names. But sometimes you could be interested in only the content with the filenames removed. You can have that if you make **grep** ignorant of the source of its input. Concatenate the files with **cat** and pipe the combined output to **grep**:

```
$ cat foo[123] | grep "print"
print "Content-type: text/html\n\n";
print "</html>\n";
find / -mtime +7 -print | perl -ne 'chop ; unlink ;'
$slno++ ; print ($slno . " " . $_ . "\n") ;
printf "File $file was last modified %0.3f days back \n", $m_age ;
```

Since **grep** acts on a stream this time, the filenames vanish from the output. You'll come across similar situations as you work your way through.

Note

In a pipeline, the command on the left of the | must use standard output, and the one on the right must use standard input.

8.9 tee: Splitting a Stream

The UNIX **tee** command breaks up its input into two components; one component is saved in a file, and the other is connected to the standard output. **tee** doesn't perform any filtering action on its input; it gives out exactly what it takes. Even though it isn't a feature of the shell, it has been included in this chapter because it involves the handling of a character stream, which this chapter is mostly about.

Being a filter (uses standard input and standard output), **tee** can be placed anywhere in a pipeline. You can use **tee** to save the output of the **who** command in a file and display it as well:

```
$ who | tee user.lst
romeo      tty01       May 18 09:32
juliet     tty02       May 18 11:18
andrew     tty03       May 18 13:21
```

You can crosscheck the display with the contents of the file `user.lst`:

```
$ cat user.lst
romeo      tty01       May 18 09:32
juliet     tty02       May 18 11:18
andrew     tty03       May 18 13:21
```

You can pipe **tee**'s output to another command, say **wc**:

```
$ who | tee user.lst | wc -l
      3
```

How do you use **tee** to display, both the list of users and its count on the terminal? Since the terminal is also a file, you can use the device name `/dev/tty` as an argument to **tee**:

```
$ who | tee /dev/tty | wc -l                  /dev/tty used as command argument
romeo  tty01           May 18 09:32
juliet tty02           May 18 11:18
andrew tty03           May 18 13:21
      3
```

The advantage of treating the terminal as a file is apparent from the above example. You couldn't have done so if **tee** (or for that matter, any UNIX command) had placed restrictions on the type of file it could handle. Here the terminal is treated in the same way as any disk file. **tee** also uses the `-a` (append) option which appends the output rather than overwrites it.

8.10 Command Substitution

The shell enables the connecting of two commands in yet another way. While a pipe connects the standard output of a command to the standard input of another, the shell enables the *argument* of a command to be obtained from the standard output of another. This feature is called **command substitution**.

To consider a simple example, suppose you wish to echo today's date with a statement like this:

```
The date today is Wed Oct 20 10:12:19 EST 1999
```

Now the last part of the statement (beginning from "Wed") represents the output of the **date** command. How does one incorporate this **date** command into the **echo** statement? With command substitution, it's a simple matter. Use the expression `` `date` `` as an argument to **echo**:

```
$ echo The date today is `date`
The date today is Wed Oct 20 10:12:19 EST 1999
```

When scanning the command line, the ` (backquote or backtick) is another metacharacter that the shell looks for. There is a special key on your keyboard (generally at the top-left) that generates this key, and should not be confused with the single quote ('). The shell then executes the enclosed command, and replaces the enclosed command text with the output of the command. For command substitution to work, the command so "backquoted" must use standard output. **date** does; that's why command substitution worked.

You can use this feature to generate useful messages. For example, you can use two commands in a pipeline, and then use the output as the argument to a third:

```
$ echo "There are `ls | wc -l` files in the current directory"
There are 58 files in the current directory
```

The command worked properly even though the arguments were double-quoted. It's a different story altogether when single quotes are used:

```
$ echo 'There are `ls | wc -l` files in the current directory'
There are `ls | wc -l` files in the current directory
```

We encounter the first difference between the use of single and double quotes. The ` is one of the few characters interpreted by the shell when placed within double quotes. If you want to echo a literal `, you have to use single quotes.

Command substitution has interesting application possibilities. It speeds up work by letting you combine a number of instructions in one. You'll see more of this feature in subsequent chapters.

Note

Command substitution is enabled when the backquotes and the enclosed command are placed within double quotes. If you use single quotes, then it's not.

KORN Shell

BASH Shell

The Korn shell and bash also offer a more readable synonym for the backquote. You can place the command inside parentheses and precede the string with a $:

```
$ echo $(date)
Mon Sep 20 20:09:23 EST 1999
```

If you are using either of these shells, then you should adopt this form rather than the unreadable and archaic form using backquotes. This is the form recommended by POSIX as well.

8.11 Shell Variables

You can define and use variables both in the command line and shell scripts. These variables are called **shell variables**. A shell variable is of the string type, which means that the value is stored in ASCII rather than in binary format. No type declaration is necessary before you can use a shell variable.

All shell variables take on the generalized form *variable*=*value*. They are assigned with the = operator but evaluated by prefixing the variable name with a $. Here's an example:

```
$ x=37                    No whitespace on either side of =
$ echo $x                 $ required only at time of evaluation
37
```

A variable name comprises the letters of the alphabet, numerals and the underscore character; the first character must be a letter. Moreover, the shell is sensitive to case; the variable `x` is different from `X`. To remove a variable, use **`unset`**:

```
$ unset x
$ echo $x                 Variable removed
$ _
```

All shell variables are initialized to null strings by default. Sometimes, you'll need to explicitly set them to null values:

```
x=          x=''          x=""
```

Caution

For assigning values to shell variables, make sure that there are no spaces on either side of the =. If you provide them, the shell will treat the variable as a command and the = and value as its arguments!

To assign multiword strings to a variable, you can escape the space character, but quoting is the preferred solution:

```
$ msg='You have mail' ; echo $msg
You have mail
```

Now that you have another special character to deal with ($) that is gobbled up by the shell, you may still need to interpret it literally without it being evaluated. This can be done by either single-quoting the expression containing the $ or by escaping the $:

```
$ echo 'The average pay is $1000'
The average pay is $1000
$ echo The average pay is \$1000
The average pay is $1000
```

The output is predictable enough, but when you enclose the arguments within double quotes, you get a different result:

```
$ echo "The average pay is $1000"
The average pay is 000
```

Here is the second difference between the use of single and double quotes. Like the backquote, the $ is also evaluated by the shell when it is double-quoted. Here, the shell evaluated a “variable” `$1`; it’s undefined, so a null string was output. `$1` belongs to a set of parameters called *positional parameters (18.4)* that signify the arguments you pass to a script.

Note

Like command substitution, variable evaluation doesn’t take place within single quotes but only within double quotes.

C Shell

The C shell uses the **set** statement to set variables. There either has to be whitespace on both sides of the = or none at all:

```
set x = 10
set mydir=`pwd`
```

The evaluation is done in the normal manner by prefixing a $ to the variable name. The C shell uses another statement, **setenv**, to set a different type of variable; you’ll meet them in Chapter 17.

8.11.1 Using Variables

Setting a Pathname to a Variable In the command line, you can set a pathname to a variable and then use its shorthand representation with the **cd** command:

```
$ mfile='/usr/spool/mail'
$ cd $mfile
$ pwd
/usr/spool/mail
```

Now, suppose you have to use this absolute pathname (`/usr/spool/mail/`) several times in a script. You can assign it to a variable at the beginning of the script and then

use it everywhere—even in other scripts run by this script. Later, you may decide to change the location of the mail directory to /var/spool/mail. For everything to work as before, you need to just change the variable definition—nothing else.

Setting a UNIX Command to a Variable A shell variable can be used to replace even the command itself. If you are using a command with specific options many times, then set this command line to a variable. Just prefix the variable with a $ to execute it:

```
$ backup="tar -cvf /dev/fd0h1440 *"          fd0 is a device name
$ $backup                                     Note how the variable is "executed"
....
```

The output you see is from the execution of the **tar** command, a UNIX utility used for backing up files. Here again, you can appreciate the advantages of defining a variable and using it everywhere. If the backup device changes, just replace fd0h1440 by the new device name in the definition.

Using Command Substitution to Set Variables You can also use the feature of command substitution to set variables. For instance, if you were to set the complete pathname of the present directory to a variable mydir, you could use

```
$ mydir=`pwd`
$ echo $mydir
/usr/romeo
```

Variable usage isn't restricted to the user alone. The UNIX system also uses a number of variables to control its behavior. There are variables that tell you the type of terminal you are using, the prompt string that you use, or the directory where the incoming mail is kept. These variables are often called **environment variables** because they can alter the operation of the environment in many ways. A detailed discussion of the significance of these special shell variables will be taken up in Chapter 17.

8.12 Shell Scripts

The shell offers the facility of storing a group of commands in a file and then executing the file. All such files are called **shell scripts**. You'll also find people referring to them as *shell programs* and *shell procedures*. The instructions stored in these files are executed in the interpretive mode—much like the batch (.BAT) files of Windows.

The following shell script has a sequence of three UNIX commands stored in a file **script.sh**. You can create the file with **vi** or **emacs**, but since this takes only three lines, you can use **cat** instead:

```
$ cat > script.sh
directory=`pwd`
echo The date today is `date`
echo The current directory is $directory
[Ctrl-d]
$ _
```

The extension `.sh` is used only for the purpose of identification; it can have any extension, or even none. Try executing the file containing these commands by simply invoking the filename:

```
$ script.sh
script.sh: execute permission denied
```

Executable permission is *usually* necessary for any shell procedure to run, and by default, a file doesn't have this permission on creation. Use **chmod** to first accord executable status to the file before executing it:

```
$ chmod u+x script.sh
$ script.sh
The date today is Thu Feb 17 11:30:53 EST 2000
The current directory is /home/sumit/project5
```

The script executes the three statements in sequence. Even though we used the shell as an interpreter, it is also a programming language. You can have all the standard constructs like **if**, **while** and **for** in a shell script. The behavior of the UNIX system is controlled by many prewritten shell scripts that are executed during system startup and those written by the system administrator. Two chapters in this text are reserved for shell programming. (Chapters 18 and 19).

8.13 The Shell's Treatment of the Command Line

Now that you have seen the major features of the shell, it's time you understood the sequence of steps that it follows when processing a command. After the command line is terminated by hitting *[Enter]*, the shell goes ahead with processing the command line in one or more passes. The sequence is well-defined and assumes the following order:

- **PARSING** The shell first breaks up the command line into words, using spaces and tabs as delimiters, unless quoted. All consecutive occurrences of a space or a tab are replaced here with a single space.
- **VARIABLE EVALUATION** All words preceded by a $ are evaluated as variables unless quoted or escaped.
- **COMMAND SUBSTITUTION** Any command surrounded by backquotes is executed by the shell, and its output inserted in the place it was found.
- **REDIRECTION** The shell then looks for the characters >, < and > to open the files that they point to.
- **WILD CARD INTERPRETATION** The shell scans the command line for wild cards and replaces them with a list of filenames that match the pattern.
- **PATH EVALUATION** It finally looks for the `PATH` variable to determine the sequence of directories it has to search in order to hunt for the command.

The preceding sequence can be considered as a simplistic treatment of the Bourne shell's behavioral pattern; the C shell has a different pattern. Remember that there are several other things that the shell looks for that have been ignored here. For example, the character **;** (as well as `||` and `&&`) stops the shell from reading any

further. There are other characters which are acted upon, and you'll come across them as the shell's features are gradually revealed. This revelation will be spread across several chapters.

8.14 The Other Shells

The original UNIX system came with the Bourne shell. Then came a plethora of shells offering new features. Two of them, the Korn shell (represented by the command file **ksh**) and the bash shell (**bash**) have been very well accepted by the UNIX fraternity. The Korn shell is offered as standard in SVR4, and bash is the standard shell in Linux. The C shell (**csh**) predates them both. It is still popular but will ultimately have to make way for Korn and bash.

Korn and bash maintain near-complete compatibility with the Bourne shell, which means that all shell programs developed under Bourne will also run under these two shells. Because commands like **kill** and integer and string handling features are built in, programs run under Korn and bash execute faster than under Bourne. In this book, the features of the C, Korn and bash shells are highlighted suitably. Further, the exclusive programming features of Korn and bash are discussed in Chapter 19. The C shell's programming constructs have been documented in Appendix A.

➤ *GOING FURTHER*

GOING FURTHER

8.15 More Wild Cards in Korn Shell and bash

At times, you'll face situations when you can't frame a single expression to match a group of filenames. For instance, how do you match the following filenames?

```
chap01 chap02 chap03 chap16 chap17 chap18 chap19
```

The Bourne shell needs to use two expressions—`chap0[1-3] chap1[6-9]`—even though there's a common string (`chap`) in both of them. bash and Korn provide an important extension to the standard wild-card set *(8.2)* by letting you enclose differing expressions within curly braces. This expression matches them all:

```
chap{0[1-3],1[6-9]}                                   Note the comma
```

The comma here acts as the delimiter between the uncommon expressions placed within the braces. There must not be any whitespace on either side. And here's how you can copy the `.txt` and `.gz` versions of the files `README` and `INSTALL`:

```
cp {INSTALL,README}.{gz,txt} ../doc
```

This feature shortens the command line considerably; with Bourne, you would have had to specify all the four filenames separately. It also means that you can access multiple directories using a shortened syntax:

```
cp /home/romeo/{project,html,scripts}/* .
```

This copies all files from the three directories (`project`, `html` and `scripts`) to the current directory. Isn't this convenient? This feature is also available in the C shell, but not the one that is discussed next.

The Invert Selection Feature If you have used Windows Explorer, you would no doubt have used the Invert Selection feature. This option reverses the selection you make with your mouse and highlights the rest. bash and Korn also provide a similar feature of matching all filenames *except* those in the expression. For instance, this expression

```
*.!(exe)                                   All files without .exe extension
```

matches all except the `.exe` files. If you want to include multiple expressions in the exception list, then use the `|` as the delimiter:

```
cp !(*.jpg|*.jpeg|*.gif) ../text
```

This copies all except the graphic files in GIF or JPEG format to the `text` directory. Note that the parentheses and `|` can be used to group filenames only if the `!` precedes the group.

Tip

The exclusion feature won't work in bash unless you make the setting **`shopt -s extglob`**. Even if you don't understand what this means, simply place this statement in `.bash_profile` or `.profile`, whichever is your startup file.

SUMMARY

The shell is a command that runs when a user logs in, and terminates when she logs out. It waits for a command to be entered and scans it for special characters (*metacharacters*). It rebuilds the command line before turning it over to the kernel for execution.

The shell matches filenames with *wild cards* that must be expanded before the command is executed. It can match any character (`*`) or a single one (`?`). It can also match a range (`[]`) and negate a match (`!`). These characters mean nothing to a command. However, **`find`** uses its own set of wild cards.

Any wild card or special character is *escaped* with a `\` to be treated literally, and if there are a number of them, then they should be placed within quotes. The `\` also escapes the *[Enter]* key, enabling you to split a lengthy command line into multiple lines.

Sometimes, escaping is used by a command to attach (rather than remove) a special meaning to a character. The **`echo`** command uses special escape sequences for echoing the formfeed character (`\f`), newline (`\n`) and tab (`\t`). **`echo`** also uses octal values. **`echo \007`** produces a beep.

Many commands use data in the form of a *character stream*. They take input from the *standard input* stream and direct output to the *standard output* stream. By default, they are set to the keyboard and terminal, respectively. They can also be redirected to come from or go to a disk file or *pipeline*.

The symbol > overwrites an existing file and >> appends to it by redirecting standard output. < redirects standard input. Commands using standard input and standard output are called *filters*, a number of which are in the UNIX system.

The *standard error* represents error messages. Its default destination is the terminal, but it can also be redirected with 2>.

The file `/dev/null` is a special file that never grows in size even when a stream of data is directed to it. `/dev/tty` is a generic device name for every terminal which every user can use to direct output to.

Using a *pipeline*, the standard output of one command can be connected to the standard input of another. A combination of filters placed in pipelines can be used to perform complex tasks which the commands can't perform individually.

The **tee** command breaks the output into two streams. One stream goes to the standard output, and the other is saved in a file. **tee** is an external UNIX command and not a feature of the shell.

Command substitution enables a command's output to become the arguments of another command. It is specified within a pair of backquotes (``).

Shell variables are used to store values that can be used in script logic. They are of the form *variable* = *value* but are evaluated by prefixing a $ to the variable name. The variables that control the workings of the UNIX system are known as *environment variables*.

Single quotes protect all special characters, while double quotes enable variable evaluation and command substitution.

The shell is also a programming language with its own set of constructs like **if**, **for** and **while**. These constructs can be used in combination with UNIX commands and variables in a *shell script*. A shell script generally requires executable permission.

The Bourne shell (`sh`) is the universal shell, though the C shell (`csh`) also has a significant user base. The Korn shell (`ksh`) and the bash shell (`bash`) are superior alternatives to the Bourne shell and C shell.

GOING FURTHER

The Korn shell and bash extend the wild-card matching features of the Bourne shell. They use the symbols {} to group multiple patterns using the **,** as the delimiter of patterns. The ! is used with the grouping operators () and the delimiter | for selecting all files except those matching an expression.

SELF-TEST

8.1 Why does the shell have to expand the wild cards?

8.2 What does the shell do when it encounters the * as a single argument to a command?

8.3 Match the filenames `chapa`, `chapb`, `chapc`, `chapx`, `chapy` and `chapz` with one expression.

8.4 Does **rm *** remove all files?

8.5 How do you list all filenames that have at least four characters?

8.6 Which UNIX command uses wild cards as part of its syntax?
8.7 When using **cat > foo**, what happens if foo already contains something?
8.8 What happens when you use **who >> foo** and foo doesn't exist?
8.9 You have a long command sequence which you want to split into multiple lines. What precautions do you need to take?
8.10 What is this command meant to do? Is it legitimate in the first place?

```
>foo <bar bc
```

8.11 What is the best method of ensuring that error messages are not seen on the terminal?
8.12 Make this setting at the command prompt. Can you execute $x?

```
x='ls | more'
```

8.13 Enter the commands **echo "$SHELL"** and **echo '$SHELL'**. What difference do you notice?
8.14 How do you find out the number of users logged in?
8.15 Attempt the variable assignment x = 10 (space on both sides of the =). Does it work?
8.16 What is the difference between directory='pwd' and directory=`pwd`?
8.17 What is the standard shell used in Linux?
8.18 The command **echo "Enter your name\c"** didn't put the cursor at the end of the prompt in Linux. Why?

EXERCISES

8.1 Using wild cards, frame a pattern where the first character is alphabetic and the last character is not numeric.
8.2 What is the significance of the command **ls *.***? Does it match files that don't contain a dot?
8.3 Consider the pattern .*.*[!.] How many dots could there be in filenames that match this pattern?
8.4 How do you remove only the hidden files of your directory?
8.5 How do you remove a file beginning with a hyphen in the foo directory if you are not using the C shell?
8.6 Is the expression [3-h]* valid?
8.7 Match all filenames not beginning with a dot.
8.8 Will **ls .*swp** show the filename .ux.2.swp if it exists?
8.9 How do you mark the completion of a command with a beep?
8.10 When does **cd *** work?
8.11 What happens when you use **cat foo > foo**?
8.12 Execute the command **ls > newlist**. What interesting observation can you make from the contents of newlist?
8.13 You want to concatenate two files, foo1 and foo2, but also insert some text in between from the terminal. How will you do this?

8.14 When will **wc < chap0[1-5]** work?

8.15 Is the output of the command **cat foo1 foo2 >/dev/tty** directed to the standard output?

8.16 What's the difference between the two lines produced by two invocations of **wc**? Why is the filename missing in the second line?

```
3       20       103 infile
3       20       103
```

8.17 What is a *filter*? Where does a filter get its input from?

8.18 What are the two consequences of using double quotes?

8.19 Using command substitution, write a command sequence which always prints the calendar of the current month.

8.20 For command substitution to work with a command, does the command have to be a filter?

8.21 A shell script **foo.sh** contains just this line—`who >/dev/tty`. Since the output of the command comes to the terminal, can you redirect the script by using `foo.sh > bar`?

GOING FURTHER

8.22 Without using a script, can you copy all files *not* having the `.bak` extension to a directory `foobar`? When will the command not work?

KEY TERMS

character class *(8.2.2)*
character stream *(8.6)*
command parsing *(8.13)*
command substitution *(8.10)*
despecializing *(8.3)*
environment variable *(8.11.1)*
escape sequence *(8.5)*
escaping *(8.3)*
file descriptor *(8.6.3)*
filter *(8.8)*
metacharacter *(8.2.1)*
pipeline *(8.8)*
quoting *(8.4)*
redirection *(8.6)*
shell *(8.1)*
shell script *(8.12)*
shell variable *(8.11)*
sleeping *(8.1)*
standard error *(8.6.3)*
standard input *(8.6.2)*
standard output *(8.6.1)*
waiting *(8.1)*
waking *(8.1)*
wild card *(8.2.1)*