Using the Shell

This book introduced you to the shell in **Chapter 4**. However, many important details were omitted in that chapter, which was aimed at helping you to get your Linux system up and running as quickly as possible. This section revisits the shell, providing you with information that will help you use the shell efficiently and effectively.

Typing Shell Commands

When typing shell commands, you have access to a mini-editor that resembles the DOSKEYS editor of MS-DOS. **Table 13-2** summarizes some useful keystroke commands interpreted by the shell. The keystroke commands let you access a list of recently executed commands, called the *history list*. To re-execute a command, you can press the Up key several times until you locate the command and then merely press **Enter** to execute the command.

Table 13-2. Useful Editing Keystrokes

Keystroke(s)	Function
Up	Move back one command in the history list.
Down	Move forward one command in the history list.
Left	Move back one character.
Right	Move forward one character.
Esc f	Move forward one word.
Esc b	Move back one word.

Keystroke(s)	Function
Ctrl-A	Move to beginning of line.
Ctrl-E	Move to end of line.
Ctrl-D	Delete current character.
Backspace	Delete previous character.
Esc d	Delete current word.
Ctrl-U	Delete from beginning of line.
Esc k	Delete to end of line.
Ctrl-Y	Retrieve last item deleted.
Esc .	Insert last word of previous command.
Ctrl-L	Clear the screen, placing the current line at the top of the screen.
Tab	Attempt to complete the current word, interpreting it as a filename, username, variable name, hostname, or command as determined by the context.
Esc ?	List the possible completions.

One of the most useful editing keystrokes, **Tab**, can also be used when typing a command. If you type the first part of a filename and press **Tab**, the shell will attempt to locate files with names matching the characters you've typed. If exactly one such file exists, the shell fills out the partially typed name with the proper characters. You can then press **Enter** to execute the command or continue typing other options and arguments. This feature, called either filename completion or command completion, makes the shell much easier to use.

In addition to keystrokes for editing the command line, the shell interprets several keystrokes that control the operation of the currently executing program. **Table 13-3** summarizes these keystrokes. For example, typing Ctrl-C generally cancels execution of a program. This keystroke command is handy, for example, when a program is taking too long to execute and you'd prefer to try something else.

Table 13-3. Useful Control Keystrokes

Keystroke	Function
Ctrl-C	Sends an interrupt signal to the currently executing command, which generally responds by terminating itself.
Ctrl-D	Sends an end of file to the currently executing command. Use this keystroke to terminate console input.
Ctrl-Z	Suspends the currently executing program.

Several other special characters control the operation of the shell, as shown in **Table 13-4**. The `#` and `;` characters are most often used in shell scripts, which you'll learn about later in this chapter. The `&` character is useful for running a command as a background process.

Table 13-4. Other Special Shell Characters

Character	Function
`#`	Marks the command as a comment, which the shell ignores.
`;`	Separates commands, letting you enter several commands on a single line.
`&`	Placed at the end of a command, causes the command to execute as a background process, so that a new shell prompt appears immediately after the command is entered.

Commands and Arguments

As you already know, the general form a shell command line is this:

> *command options arguments*

The command determines what operation the shell will perform and the options and arguments customize or fine-tune the operation. Sometimes the command specifies a program file that will be launched and run; such a command is called an *external command*. Linux generally stores these files in `/bin`, `/usr/bin`, or `/usr/local/bin`. System administration commands are generally stored in `/sbin` or `/usr/sbin`. When a command specifies a program file, the shell passes any specified arguments to the program, which scans them and interprets them, adjusting its operation accordingly.

However, some commands are not program files; instead they are built-in commands interpreted by the shell itself. One important way in which shells differ is the built-in commands that they support. Later in this section, you'll learn about some commands built into the BASH shell.

Filename Globbing

Before the shell passes arguments to an external command or interprets a built-in command, it scans the command line for certain special characters and performs an operation known as *filename globbing*. Filename globbing resembles the processing of wildcards used in MS-DOS commands, but it's much more sophisticated. **Table 13-5** describes the special characters used in filename globbing, known as *filename metacharacters*.

Table 13-5. Filename Metacharacters

Metacharacter	Meaning
`*`	Matches a string of zero or more characters
`?`	Matches exactly one character

Metacharacter	Meaning
`[ abc ...]`	Matches any of the characters specified
`[ a - z ]`	Matches any character in the specified range
`[! abc ...]`	Matches any character other than those specified
`[! a - z ]`	Matches any character not in the specified range
`~`	The home directory of the current user
`~ userid`	The home directory of the specified user
`~+`	The current working directory
`~-`	The previous working directory

In filename globbing just as in MS-DOS wildcarding, the shell attempts to replace metacharacters appearing in arguments in such a way that arguments specify file-names. Filename globbing makes it easier to specify names of files and sets of files.

For example, suppose the current working directory contains the following files: `file1`, `file2`, `file3`, and `file04`. Suppose you want to know the size of each file. The following command reports that information:

```
ls -l file1 file2 file3 file04
```

However, the following command reports the same information and is much easier to type:

```
ls -l file*
```

As **Table 13-2** shows, the `*` filename metacharacter can match any string of characters. Suppose you issued the following command:

```
ls -l file?
```

The `?` filename metacharacter can match only a single character. Therefore, `file04` would not appear in the output of the command.

Similarly, the command:

```
ls -l file[2-3]
```

would report only `file2` and `file3`, because only these files have names that match the specified pattern, which requires that the last character of the filename be in the range `2-3`.

You can use more than one metacharacter in a single argument. For example, consider the following command:

```
ls -l file??
```

This command will list `file04`, because each metacharacter matches exactly one filename character.

Most commands let you specify multiple arguments. If no files match a given argument, the command ignores the argument. Here's another command that reports all four files:

```
ls -l file0* file[1-3]
```

TIP

Suppose that a command has one or more arguments that include one or more metacharacters. If none of the arguments matches any filenames, the shell passes the arguments to the program with the metacharacters intact. When the program expects a valid filename, an unexpected error may result.

Another metacharacter lets you easily refer to your home directory. For example, the following command:

```
ls ~
```

lists the files in the user's home directory.

Filename metacharacters don't merely save you typing. They let you write scripts that selectively process files by name. You'll see how that works later in this chapter.

Shell Aliases

Shell aliases make it easier to use commands by letting you establish abbreviated command names and by letting you pre-specify common arguments. To establish a command alias, issue a command of the form:

```
alias name='command'
```

where `command` specifies the command for which you want to create an alias and `name` specifies the alias. For example, suppose you frequently type the MS-DOS command **Dir** when you intend to type the Linux command **ls**. You can establish an alias for the **ls** command by issuing this command:

```
alias dir='ls -l'
```

Once the alias is established, if you mistakenly type **Dir**, you'll nevertheless get the directory listing you want. If you like, you can establish similar aliases for other commands.

Your default Linux configuration probably defines several aliases on your behalf. To see what they are, issue the command:

```
alias
```

If you're logged in as `root`, you may see the following aliases:

```
alias cp='cp -i'
alias dir='ls -l'
alias ls='ls --color'
alias mv='mv -i'
alias rm='rm -i'
```

Notice how several commands are self-aliased. For example, the command **rm –i** is aliased as **rm**. The effect is that the *–i* option appears whenever you issue the **rm** command, whether or not you type the option. The *–i* option specifies that the shell will prompt for confirmation before deleting files. This helps avoid accidental deletion of files, which can be particularly hazardous when you're logged in as `root`. The alias ensures that you're prompted for confirmation even if you don't ask to be prompted. If you don't want to be prompted, you can issue a command like:

```
rm -f files
```

where `files` specifies the files to be deleted. The *–f* option has an effect opposite that of the *–i* option; it forces deletion of files without prompting for confirmation. Because the command is aliased, the command actually executed is:

```
rm -i -f files
```

The *–f* option takes precedence over the *–i* option, because it occurs later in the command line.

If you want to remove a command alias, you can issue the **unalias** command:

```
unalias alias
```

where `alias` specifies the alias you want to remove. Aliases last only for the duration of a log in session, so you needn't bother to remove them before logging off. If you want an alias to be effective each time you log in, you can use a shell script. The next subsection shows you how to do so.

Shell Scripts

A shell script is simply a file that contains commands. By storing commands as a shell script you make it easy to execute them again and again. As an example, consider a file named `deleter`, which contains the following lines:

```
echo -n Deleting the temporary files...
rm -f *.tmp
echo Done.
```

The **echo** commands simply print text on the console. The *–n* option of the first **echo** command causes omission of the trailing newline character normally written by the **echo** command, so both **echo** commands write their text on a single line. The **rm** command removes from the current working directory all files having names ending in `.tmp`.

You can execute this script by issuing the **sh** command:

```
sh deleter
```

TIP

If you invoke the **sh** command without an argument specifying a script file, a new interactive shell is launched. To exit the new shell and return to your previous session, issue the **exit** command.

If the `deleter` file were in a directory other than the current working directory, you'd have to type an absolute path, for example:

```
sh /home/bill/deleter
```

You can make it a bit easier to execute the script by changing its access mode to include execute access. To do so, issue the following command:

```
chmod ugo+x deleter
```

This gives you, members of your group, and everyone else the ability to execute the file. To do so, simply type the absolute path of the file, for example:

```
/home/bill/deleter
```

If the file is in the current directory, you can issue the following command:

```
./deleter
```

You may wonder why you can't simply issue the command:

```
deleter
```

In fact, this still simpler form of the command will work, so long as `deleter` resides in a directory on your search path. You'll learn about the search path later.

Linux includes several standard scripts that are run at various times. **Table 13-6** identifies these and gives the time when each is run. You can modify these scripts to operate differently. For example, if you want to establish command aliases that are available whenever you log in, you can use a text editor to add the appropriate lines to the `.profile` file that resides in your home directory. Recall that, since the name of this file begins with a dot, the **ls** command won't normally show the file. You must specify the *-a* option in order to see this and other hidden files.

Table 13-6. Special Scripts

Script	Function
`/etc/profile`	Executed when the user logs in
`~/.profile`	Executed when the user logs in

Script	Function
~/.bashrc	Executed when BASH is launched
~/.bash_logout	Executed when the user logs out

TIP

If you want to modify one of the standard scripts that should reside in your home directory, but find that your home directory does not contain the indicated file, simply create the file. The next time you log in, log out, or launch BASH (as appropriate) the shell will execute your script.

Input/Output Redirection and Piping

The shell provides three standard data streams:

`stdin`
The standard input stream

`stdout`
The standard output stream

`stderr`
The standard error stream

By default, most programs read their input from `stdin` and write their output to `stdout`. Because both streams are normally associated with a console, programs behave as you generally want, reading input data from the console keyboard and writing output to the console screen. When a well-behaved program writes an error message, it writes the message to the `stderr` stream, which is also associated with the console by default. Having separate streams for output and error messages presents an important opportunity, as you'll see in a moment.

Although the shell associates the three standard input/output streams with the console by default, you can specify input/output redirectors that, for example, associate

an input or output stream with a file. **Table 13-7** summarizes the most important input/output redirectors.

Table 13-7. Input/Output Redirectors

Redirector	Function
`>` *file*	Redirects standard output stream to specified file
`2>` *file*	Redirects standard error stream to specified file
`>>` *file*	Redirects standard output stream to specified file, appending output to the file if the file already exists
`2>>` *file*	Redirects standard error stream to specified file, appending output to the file if the file already exists
`&>` *file*	Redirects standard output and error streams to the specified file
`<` *file*	Redirects standard input stream to the specified file
`<<` *text*	Reads standard input until a line matching *text* is found, at which point end of file is posted
cmd1 `|` *cmd2*	Takes the standard input of *cmd2* from the standard output of *cmd1* (also known as the *pipe redirector*)

To see how redirection works, consider the **wc** command. This command takes a series of filenames as arguments and prints the total number of lines, words, and characters present in the specified files. For example, the command:

```
wc /etc/passwd
```

might produce the output:

```
  22    26    790 /etc/passwd
```

which indicates that the file `/etc/passwd` contains 22 lines, 26 words, and 790 characters. Generally, the output of the command appears on console. But, consider the following command, which includes an output redirector:

```
wc /etc/passwd > total
```

If you issue this command, you'll see no console output, because the output is redirected to the file `total`, which the command creates (or overwrites, if the file already exists). If you execute the pair of commands:

```
wc /etc/passwd > total
cat total
```

you can see the output of the **wc** command on the console.

Perhaps you can now see the reason for having the separate output streams `stdout` and `stderr`. If the shell provided a single output stream, error messages and output would be mingled. Therefore, if you redirected the output of a program to a file, any error messages would also be redirected to the file. This might make it difficult to notice an error that occurred during program execution. Instead, because the streams are separate, you can choose to redirect only `stdout` to a file. When you do so, error messages sent to `stderr` appear on the console in the usual way. Of course, if you prefer, you can redirect both `stdout` and `stderr` to the same file or redirect them to different files. As usual in the Unix world, you can have it your own way.

A simple way of avoiding annoying output is to redirect it to the null file, `/dev/null`. If you redirect the `stderr` stream of a command to `/dev/null`, you won't see any error messages the command produces.

Just as you can direct the standard output or error stream of a command to a file, you can also redirect a command's standard input stream to a file, so that the command reads from the file instead of the console. For example, if you issue the **wc** command without arguments, the command reads its input from `stdin`. Type some words and then type the end of file character (Ctrl-D) and **wc** will report the number of lines,

words, and characters you entered. You can tell **wc** to read from a file, rather than the console, by issuing a command like:

```
wc </etc/passwd
```

Of course, this isn't the usual way of invoking **wc**. The author of **wc** helpfully provided a command-line argument that lets you specify the file from which **wc** reads. However, by using a redirector, you could read from any desired file even if the author had been less helpful.

TIP

Some programs are written to ignore redirectors. For example, the **passwd** command expects to read the new password only from the console, not from a file. You can compel such programs to read from a file, but doing so requires techniques more advanced than redirectors.

When you specify no command-line arguments, many Unix programs read their input from `stdin` and write their output to `stdout`. Such programs are called *filters*. Filters can be easily fitted together to perform a series of related operations. The tool for combining filters is the *pipe*, which connects the output of one program to the input of another. For example, consider this command:

```
ls -l ~ | wc -l
```

The command consists of two commands, joined by the pipe redirector (`|`). The first command lists the names of the files in the users home directory, one file per line. The second command invokes **wc** by using the *-l* option, which causes **wc** to print only the total number of lines, rather than printing the total number of lines, words, and characters. The pipe redirector sends the output of the **ls** command to the **wc** command, which counts and prints the number of lines in its input, which happens to be the number of files in the user's home directory.

This is a simple example of the power and sophistication of the Unix shell. Unix doesn't include a command that counts the files in the user's home directory and

doesn't need to do so. Should the need to count the files arise, a knowledgeable Unix user can prepare a simple script that computes the desired result by using general-purpose Unix commands.

Shell Variables

If you've studied programming, you know that programming languages resemble algebra. Both programming languages and algebra let you refer to a value by a name. And both programming languages and algebra include elaborate mechanisms for manipulating named values.

The shell is a programming language in its own right, letting you refer to variables known as *shell variables* or *environment variables*. To assign a value to a shell variable, you use a command that has the following form:

```
variable=value
```

For example, the command:

```
DifficultyLevel=1
```

assigns the value `1` to the shell variable named `DifficultyLevel`. Unlike algebraic variable, shell variables can have non-numeric values. For example, the command:

```
Difficulty=medium
```

assigns the value `medium` to the shell variable named `Difficulty`.

Shell variables are widely used within Unix, because they provide a convenient way of transferring values from one command to another. Programs can obtain the value of a shell variable and use the value to modify their operation, in much the same way they use the value of command-line arguments.

You can see a list of shell variables by issuing the **set** command. Usually, the command produces more than a single screen of output. So, you can use a pipe redirector and the **more** command to view the output one screen at a time:

```
set | more
```

Press the **Space** bar to see each successive page of output. You'll probably see several of the shell variables described in **Table 13-8** among those printed by the **set** command. The values of these shell variables are generally set by one or another of the startup scripts described earlier in this chapter.

Table 13-8. Important Environment Variables

Variable	Function
DISPLAY	The X display to be used; for example, `localhost:0`
HOME	The absolute path of the user's home directory
HOSTNAME	The Internet name of the host
LOGNAME	The user's login name
MAIL	The absolute path of the user's mail file
PATH	The search path (see next subsection)
SHELL	The absolute path of the current shell
TERM	The terminal type
USER	The user's current username; may differ from the login name if the user executes the **su** command

You can use the value of a shell variable in a command by preceding the name of the shell variable by a dollar sign ($). To avoid confusion with surrounding text, you can enclose the name of the shell variable within curly braces ({}); it's good practice (though not necessary) to do so consistently. For example, you can change the current working directory to your home directory by issuing the command:

```
cd ${HOME}
```

Of course, issuing the **cd** command with no argument causes the same result. However, suppose you want to change to the `work` subdirectory of your home directory. The following command accomplishes exactly that:

```
cd ${HOME}/work
```

An easy way to see the value of a shell variable is to specify the variable as the argument of the **echo** command. For example, to see the value of the `HOME` shell variable, issue the command:

```
echo ${HOME}
```

To make the value of a shell variable available not just to the shell, but to programs invoked by using the shell, you must export the shell variable. To do so, use the **export** command, which has the form:

```
export variable
```

where *variable* specifies the name of the variable to be exported. A shorthand form of the command lets you assign a value to a shell variable and export the variable in a single command:

```
export variable=value
```

You can remove the value associated with shell variable by giving the variable an empty value:

```
variable=
```

However, a shell variable with an empty value remains a shell variable and appears in the output of the **set** command. To dispense with a shell variable, you can issue the **unset** command:

```
unset variable
```

Once you unset the value of a variable, the variable no longer appears in the output of the **set** command.

The Search Path

The special shell variable `PATH` holds a series of paths known collectively as the *search path*. Whenever you issue an external command, the shell searches paths that comprise the search path, seeking the program file that corresponds to the command. The startup scripts establish the initial value of the `PATH` shell variable, but you can modify its value to include any desired series of paths. You must use a colon (:) to separate each path of the search path.

For example, suppose that PATH has the following value:

```
/usr/bin:/bin:/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin
```

You can add a new search directory, say `/home/bill`, with the following command:

```
PATH=${PATH}:/home/bill
```

Now, the shell will look for external programs in `/home/bill` as well as the default directories. However, it will look there last. If you prefer to check `/home/bill` first, issue the following command instead:

```
PATH=/home/bill:${PATH}
```

The **which** command helps you work with the `PATH` shell variable. It checks the search path for the file specified as its argument and prints the name of the matching path, if any. For example, suppose you want to know where the program file for the **wc** command resides. Issuing the command:

```
which wc
```

will tell you that the program file is `/usr/bin/wc`, or whatever other path is correct for your system.

Quoted Strings

Sometimes the shell may misinterpret a command that you've written, globbing a filename or expanding a reference to a shell variable that you hadn't intended. Of course, it's actually your interpretation that's mistaken, not the shell's. Therefore, it's up to you to rewrite your command so that the shell's interpretation is congruent with what you intend.

Quote characters, described in **Table 13-9**, can help you do so, by controlling the operation of the shell. For example, by enclosing a command argument within single quotes, you can prevent the shell from globbing the argument or substituting the argument with the value of a shell variable.

Table 13-9. Quote Characters

Character	Function
`'`	Characters within a pair of single quotes are interpreted literally; that is, their metacharacter meanings (if any) are ignored. Similarly, the shell does not replace references to shell or environment variables with the value of the referenced variable.
`"`	Characters within a pair of double quotes are interpreted literally; that is, their metacharacter meanings (if any) are ignored. However, the shell does replace references to shell or environment variables with the value of the referenced variable.
`` ` ``	Text within a pair of back quotes is interpreted as a command, which the shell executes before executing the rest of the command line. The output of the command replaces the original back-quoted text.
`\`	The following character is interpreted literally; that is, its metacharacter meaning (if any) is ignored. The backslash character has a special use as a line continuation character. When a line

Character	Function
	ends with a backslash, the line and the following line are considered part of a single line.

To see this in action, consider how you might cause the **echo** command to produce the output `$PATH`. If you simply issue the command:

```
echo $PATH
```

the echo command will print the value of the `PATH` shell variable. However, by enclosing the argument within single quotes, you obtain the desired result:

```
echo '$PATH'
```

Double quotes have a similar effect. They prevent the shell from globbing a filename but permit the expansion of shell variables.

Back quotes operate differently; they let you execute a command and use its output as an argument of another command. For example, the command:

```
echo My home directory contains `ls ~ | wc -l` files.
```

prints a message that gives the number of files in the user's home directory. The command works by first executing the command contained within back quotes:

```
ls ~ | wc -l
```

This command, as explained earlier, computes and prints the number of files in the user's directory. Because the command is enclosed in back quotes, its output is not printed; instead the output replaces the original back quoted text.

The resulting command becomes:

```
echo My home directory contains 22 files.
```

When executed, this command prints the output:

```
My home directory contains 22 files.
```

The Power of the Linux Shell

You may now begin to appreciate the power of the Linux shell: by including command aliases in your *bashrc* script, you can extend the command repertoire of the shell. And, by using filename completion and the history list, you can reduce the amount of typing necessary. Once you grasp how to properly use it, the Linux shell is a powerful, fast, and easy to use interface that avoids the limitations and monotony of the more familiar point-and-click graphical interface.

But, the shell has additional features that extend its capabilities even further. As you'll see in the next section, the Linux shell includes a powerful programming language that provides argument processing, conditional logic, and loops.

Previous chapter
< 13. Conquering the BASH Shell

Next chapter
Understanding Shell Scripts >