chapter 7

Working with the BASH Shell

After completing this chapter, you will be able to:

- Redirect the input and output of a command
- Identify and manipulate common shell environment variables
- Create and export new shell variables
- Edit environment files to create variables upon shell startup
- Describe the purpose and nature of shell scripts
- Create and execute basic shell scripts
- Effectively use common decision constructs in shell scripts

A solid understanding of shell features is vital to both administrators and users, who must interact with the shell on a daily basis. The first part of this chapter describes how the shell can manipulate command input and output using redirection and pipe shell metacharacters. Next, you explore the different types of variables present in a BASH shell after login, as well as their purpose and usage. Finally, this chapter ends with an introduction to creating and executing BASH shell scripts.

Command Input and Output

The BASH shell is responsible for providing a user interface and interpreting commands entered on the command line. In addition, the BASH shell can manipulate command input and output, provided the user specifies certain shell metacharacters on the command line alongside the command. Command input and output are represented by labels known as **file descriptors**. For each command that can be manipulated by the BASH shell, there are three file descriptors:

- Standard Input (stdin)
- Standard Output (stdout)
- Standard Error (stderr)

Standard Input (**stdin**) refers to the information processed by the command during execution; this often takes the form of user input typed on the keyboard. **Standard Output** (**stdout**) refers to the normal output of a command, whereas **Standard Error** (**stderr**) refers to any error messages generated by the command. Both Standard Output and Standard Error are displayed on the terminal screen by default. All three components are depicted in Figure 7-1.

As shown in Figure 7-1, each file descriptor is represented by a number, with stdin represented by the number 0, stdout represented by the number 1, and stderr represented by the number 2.

Although all three descriptors are available to any command, not all commands use every descriptor. The `ls /etc/hosts /etc/h` command used in Figure 7-1 gives Standard

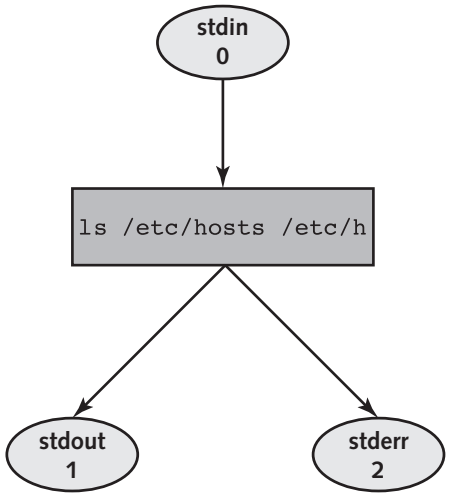

Figure 7-1 The three common file descriptors

Output (the listing of the /etc/hosts file) and Standard Error (an error message indicating that the /etc/h file does not exist) to the terminal screen, as shown in the following output:

```
[root@server1 ~]# ls /etc/hosts /etc/h
ls: cannot access /etc/h: No such file or directory
/etc/hosts
[root@server1 ~]# _
```

Redirection

You can use the BASH shell to redirect Standard Output and Standard Error from the terminal screen to a file on the filesystem. To do this, include the > shell metacharacter followed by the absolute or relative pathname of the file. For example, to redirect only the Standard Output to a file called goodoutput for the command used in Figure 7-1, you append the number of the file descriptor (1) followed by the **redirection** symbol > and the file to redirect the Standard Output to (goodoutput), as shown in the following output:

```
[root@server1 ~]# ls /etc/hosts /etc/h 1>goodoutput
ls: cannot access /etc/h: No such file or directory
[root@server1 ~]# _
```

You can include a space character after the > metacharacter, but it is not necessary.

In the preceding output, the Standard Error is still displayed to the terminal screen because it was not redirected to a file. The listing of /etc/hosts was not displayed, however; instead, it was redirected to a file called goodoutput in the current directory. If the goodoutput file did not exist prior to running the command in the preceding output, it is created automatically. However, if the goodoutput file did exist prior to the redirection, the BASH shell clears its contents before executing the command. To see that the Standard Output was redirected to the goodoutput file, you can run the following commands:

```
[root@server1 ~]# ls -F
Desktop/  goodoutput
[root@server1 ~]# cat goodoutput
/etc/hosts
[root@server1 ~]# _
```

Similarly, you can redirect the Standard Error of a command to a file; simply specify file descriptor number 2, as shown in the following output:

```
[root@server1 ~]# ls /etc/hosts /etc/h 2>badoutput
/etc/hosts
[root@server1 ~]# cat badoutput
ls: cannot access /etc/h: No such file or directory
[root@server1 ~]# _
```

In the preceding output, only the Standard Error was redirected to a file called badoutput; thus, the Standard Output (a listing of /etc/hosts) was displayed on the terminal screen.

Because redirecting the Standard Output to a file for later use is more common than redirecting the Standard Error to a file, the BASH shell assumes Standard Output in the absence of a numeric file descriptor:

```
[root@server1 ~]# ls /etc/hosts /etc/h >goodoutput
ls: cannot access /etc/h: No such file or directory
[root@server1 ~]# cat goodoutput
/etc/hosts
[root@server1 ~]# _
```

In addition, you can redirect both Standard Output and Standard Error to separate files at the same time, as shown in the following output:

```
[root@server1 ~]# ls /etc/hosts /etc/h >goodoutput 2>badoutput
[root@server1 ~]# cat goodoutput
/etc/hosts
[root@server1 ~]# cat badoutput
ls: cannot access /etc/h: No such file or directory
[root@server1 ~]# _
```

The order of redirection on the command line does not matter; the command `ls /etc/hosts /etc/h >goodoutput 2>badoutput` is the same as `ls /etc/hosts /etc/h 2>badoutput >goodoutput`.

It is important to use separate filenames to hold the contents of Standard Output and Standard Error; using the same filename for both results in a loss of data because the system attempts to write both contents to the file at the same time:

```
[root@server1 ~]# ls /etc/hosts /etc/h >goodoutput 2>goodoutput
[root@server1 ~]# cat goodoutput
/etc/hosts
access /etc/h: No such file or directory
[root@server1 ~]# _
```

To redirect both Standard Output and Standard Error to the same file without any loss of data, you must use special notation. To specify that Standard Output be sent to the file goodoutput and Standard Error be sent to the same place as Standard Output, you can do the following:

```
[root@server1 ~]# ls /etc/hosts /etc/h >goodoutput 2>&1
[root@server1 ~]# cat goodoutput
ls: cannot access /etc/h: No such file or directory
/etc/hosts
[root@server1 ~]# _
```

Alternatively, you can specify that the Standard Error be sent to the file badoutput and Standard Output be sent to the same place as Standard Error:

```
[root@server1 ~]# ls /etc/hosts /etc/h 2>badoutput >&2
[root@server1 ~]# cat badoutput
ls: cannot access /etc/h: No such file or directory
/etc/hosts
[root@server1 ~]# _
```

In all of the examples used earlier, the contents of the files used to store the output from commands were cleared prior to use by the BASH shell. Another example of this is shown in the following output when redirecting the Standard Output of the `date` command to the file dateoutput:

```
[root@server1 ~]# date >dateoutput
[root@server1 ~]# cat dateoutput
Fri Aug 20 07:54:00 EDT 2015
[root@server1 ~]# date >dateoutput
[root@server1 ~]# cat dateoutput
Fri Aug 20 07:54:00 EDT 2015
[root@server1 ~]# _
```

To prevent the file from being cleared by the BASH shell and append output to the existing output, you can specify two > metacharacters alongside the file descriptor, as shown in the following output:

```
[root@server1 ~]# date >dateoutput
[root@server1 ~]# cat dateoutput
Fri Aug 20 07:54:32 EDT 2015
[root@server1 ~]# date >>dateoutput
[root@server1 ~]# cat dateoutput
Fri Aug 20 07:54:32 EDT 2015
Fri Aug 20 07:54:48 EDT 2015
[root@server1 ~]# _
```

You can also redirect a file to the Standard Input of a command using the < metacharacter. Because there is only one file descriptor for input, there is no need to specify the number 0 before the < metacharacter to indicate Standard Input, as shown next:

```
[root@server1 ~]# cat </etc/issue
Fedora release 20 (Heisenbug)
Kernel \r on an \m (\l)

[root@server1 ~]# _
```

In the preceding output, the BASH shell located and sent the /etc/issue file to the `cat` command as Standard Input. Because the `cat` command normally takes the filename to be displayed as an argument on the command line (e.g., `cat /etc/issue`), there is no need to use Standard Input redirection with the `cat` command as used in the previous example; however, some commands on the Linux system only accept files when they are passed by the shell through Standard Input. The **`tr` command** is one such command that can be used to replace characters in a file sent via Standard Input. To translate all of the lowercase r characters in the /etc/issue file to uppercase R characters, you can run the following command:

```
[root@server1 ~]# tr r R </etc/issue
FedoRa Release 20 (Heisenbug)
KeRnel \R on an \m (\l)

[root@server1 ~]# _
```

The preceding command does not modify the /etc/issue file; it simply takes a copy of the /etc/issue file, manipulates it, and then sends the Standard Output to the terminal screen. To save a copy of the Standard Output for later use, you can use both Standard Input and Standard Output redirection together:

```
[root@server1 ~]# tr r R </etc/issue >newissue
[root@server1 ~]# cat newissue
FedoRa Release 20 (Heisenbug)
KeRnel \R on an \m (\l)

[root@server1 ~]# _
```

As with redirecting Standard Output and Standard Error in the same command, you should use different filenames when redirecting Standard Input and Standard Output. However, this is because the BASH shell clears a file that already exists before performing the redirection. An example of this is shown in the following output:

```
[root@server1 ~]# sort <newissue >newissue
[root@server1 ~]# cat newissue
[root@server1 ~]# _
```

The newissue file has no contents when displayed in the preceding output. This is because the BASH shell saw that output redirection was indicated on the command line, it cleared the contents of the file newissue, then sorted the blank file and saved the output (nothing in our example) into the file newissue. Because of this feature of shell redirection, Linux administrators commonly use the command `>filename` at the command prompt to clear the contents of a file.

The contents of log files are typically cleared periodically using the command > `/path/to/logfile`.

Table 7-1 summarizes the different types of redirection discussed in this section.

Pipes

Note from Table 7-1 that redirection only occurs between a command and a file and vice versa. However, you can send the Standard Output of one command to another command as Standard Input. To do this, you use the pipe | shell metacharacter and specify commands on either side. The shell then sends the Standard Output of the command on the left to the command on the right, which then interprets the information as Standard Input. This process is depicted in Figure 7-2.

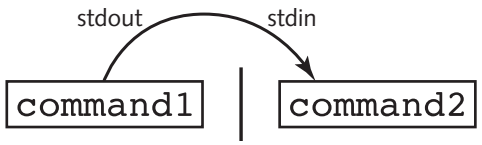

Figure 7-2 Piping information from one command to another

Command	Description
`command 1>file` `command >file`	The Standard Output of the command is sent to a file instead of to the terminal screen.
`command 2>file`	The Standard Error of the command is sent to a file instead of to the terminal screen.
`command 1>fileA 2>fileB` `command >fileA 2>fileB`	The Standard Output of the command is sent to fileA instead of to the terminal screen, and the Standard Error of the command is sent to fileB instead of to the terminal screen.

`command 1>file 2>&1`
`command >file 2>&1`
`command 1>&2 2>file command >&2 2>file`
	Both the Standard Output and the Standard Error are sent to the same file instead of to the terminal screen.

`command 1>>file` `command >>file`	The Standard Output of the command is appended to a file instead of being sent to the terminal screen.
`command 2>>file`	The Standard Error of the command is appended to a file instead of being sent to the terminal screen.
`command 0<file` `command <file`	The Standard Input of a command is taken from a file.

Table 7-1 Common redirection examples

A series of commands that includes the pipe | metacharacter is commonly referred to as a **pipe**.

The pipe symbol can be created on most keyboards by pressing Shift+\.

For example, the Standard Output of the `ls -l /etc` command is too large to fit on one terminal screen. To send the Standard Output of this command to the `less` command, which views Standard Input page-by-page, you could use the following command:

```
[root@server1 ~]# ls -l /etc | less
total 1836
drwxr-xr-x  3  root  root  4096  Dec  11   2013  abrt
-rw-r--r--  1  root  root    16  Sep  16  21:54  adjtime
drwxr-xr-x  2  root  root  4096  Sep  17  19:44  akonadi
-rw-r--r--  1  root  root  1518  Jun   7   2013  aliases
drwxr-xr-x  2  root  root  4096  Dec  11   2013  alsa
drwxr-xr-x  2  root  root  4096  Sep  24  10:06  alternatives
-rw-r--r--  1  root  root   541  Sep  25   2013  anacrontab
-rw-r--r--  1  root  root    55  Aug   1   2013  asound.conf
```

```
-rw-r--r--  1  root   root      1  Aug   2   2013  at.deny
drwxr-xr-x  2  root   root   4096  Dec  11   2013  at-spi2
drwxr-x--   3  root   root   4096  Dec  11   2013  audisp
drwxr-x--   3  root   root   4096  Dec  11   2013  audit
drwxr-xr-x  4  root   root   4096  Dec  11   2013  avahi
drwxr-xr-x  2  root   root   4096  Sep  17  19:36  bash_completion.d
-rw-r--r--  1  root   root   2815  Jun   7   2013  bashrc
drwxr-xr-x  2  root   root   4096  Dec   5   2013  binfmt.d
-rw-r-----  1  root   brlapi   33  Dec  11   2013  brlapi.key
drwxr-xr-x  2  root   root  12288  Dec  11   2013  brltty
-rw-r-r--   1  root   root  21929  Sep  23   2013  brltty.conf
-rw-r-r--   1  root   root    520  Aug   3   2013  cagibid.conf
drwxr-xr-x  2  root   root   4096  Aug   3   2013  chkconfig.d
-rw-r-r--   1  root   root   1165  Aug   9   2013  chrony.conf
:
```

You need not have spaces around the | metacharacter; the commands `ls -l/etc|less` and `ls -l/etc|less` are equivalent.

A common use of piping is to reduce the amount of information displayed on the terminal screen from commands that display too much information. Take the following output from the `mount` command:

```
[root@server1 ~]# mount
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
devtmpfs on /dev type devtmpfs (rw,nosuid,seclabel,size=1015268k,
nr_inodes=253817,mode=755)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,
noexec,relatime)
selinuxfs on /sys/fs/selinux type selinuxfs (rw,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,seclabel)
devpts on /dev/pts type devpts
(rw,nosuid,noexec,relatime,seclabel,gid=5, mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,nodev,seclabel,mode=755)
tmpfs on /sys/fs/cgroup type tmpfs
(rw,nosuid,nodev,noexec,seclabel, mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/usr/lib/systemd/
systemd-cgroups-agent,name=systemd)
pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/cpuset type cgroup
(rw,nosuid,nodev,noexec, relatime,cpuset)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,cpuacct,cpu)
cgroup on /sys/fs/cgroup/memory type cgroup
(rw,nosuid,nodev,noexec, relatime,memory)
cgroup on /sys/fs/cgroup/devices type cgroup
(rw,nosuid,nodev,noexec,relatime,devices)
```

```
cgroup on /sys/fs/cgroup/freezer type cgroup
(rw,nosuid,nodev,noexec, relatime,freezer)
cgroup on /sys/fs/cgroup/net_cls type cgroup
(rw,nosuid,nodev,noexec, relatime,net_cls)
cgroup on /sys/fs/cgroup/blkio type cgroup
(rw,nosuid,nodev,noexec, relatime,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup
(rw,nosuid,nodev,noexec, relatime,perf_event)
cgroup on /sys/fs/cgroup/hugetlb type cgroup
(rw,nosuid,nodev,noexec,relatime,hugetlb)
/dev/sda3 on / type ext4 (rw,relatime,seclabel,data=ordered)
systemd-1 on /proc/sys/fs/binfmt_misc type autofs
(rw,relatime,fd=35,
pgrp=1,timeout=300,minproto=5,maxproto=5,direct)
mqueue on /dev/mqueue type mqueue (rw,relatime,seclabel)
configfs on /sys/kernel/config type configfs (rw,relatime)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
tmpfs on /tmp type tmpfs (rw,seclabel)
/dev/sda1 on /boot type ext4 (rw,relatime,seclabel,data=ordered)
[root@server1 ~]# _
```


To view only those lines regarding ext4 filesystems, you could send the Standard Output of the `mount` command to the `grep` command as Standard Input, as shown in the following output:

```
[root@server1 ~]# mount | grep ext4
/dev/sda3 on / type ext4 (rw,relatime,seclabel,data=ordered)
/dev/sda1 on /boot type ext4 (rw,relatime,seclabel,data=ordered)
[root@server1 ~]# _
```

The `grep` command in the preceding output receives the full output from the `mount` command and then displays only those lines that have ext4 in them. The `grep` command normally takes two arguments; the first specifies the text to search for and the second specifies the filename(s) to search within. The `grep` command used in the preceding output requires no second argument because the material to search comes from Standard Input (the `mount` command) instead of from a file.

Furthermore, you can use more than one pipe | metacharacter on the command line to pipe information from one command to another command, in much the same fashion as an assembly line in a factory. Typically, an assembly line goes through several departments, each of which performs a specialized task very well. For example, one department might assemble the product, another might paint the product, and yet another might package the product. Every product must pass through each department to be complete.

You can use Linux commands that manipulate data in the same way, connecting them into an assembly line via piping. Information is manipulated by one command, and then that manipulated information is sent to another command, which manipulates it further. The process continues until the information attains the form required by the user. The piping process is depicted in Figure 7-3.

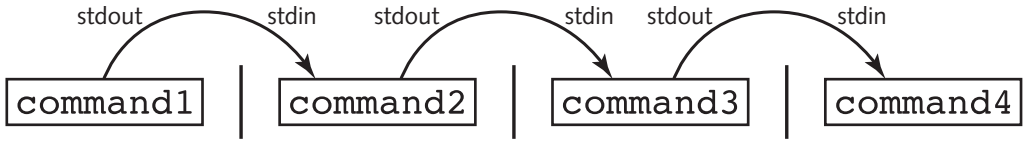

Figure 7-3 Piping several commands

Any command that can take Standard Input and transform it into Standard Output is called a **filter command**. It is important to note that commands such as `ls` and `mount` are not filter commands because they do not accept Standard Input from other commands, but instead find information from the system and display it to the user. As a result, these commands must be at the beginning of a pipe. Other commands, such as `vi`, are interactive and as such cannot exist between two pipe symbols because they cannot take from Standard Input and give to Standard Output.

Several hundred filter commands are available to Linux users. Table 7-2 lists some common ones used throughout this book.

Take, for example, the prologue from Shakespeare's *Romeo and Juliet*:

```
[root@server1 ~]# cat prologue
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
```

Command	Description
`sort` `sort -r`	Sorts lines in a file alphanumerically Reverse-sorts lines in a file alphanumerically

`wc`
`wc -l`
`wc -w`
`wc -c`
	Counts the number of lines, words, and characters in a file
	Counts the number of lines in a file
	Counts the number of words in a file
	Counts the number of characters in a file

`pr`	Formats a file for printing, with several options available; it places a date and page number at the top of each page
`pr -d`	Formats a file double-spaced
`tr`	Replaces characters in the text of a file
`grep`	Displays lines in a file that match a regular expression
`nl`	Numbers lines in a file
`awk`	Extracts, manipulates, and formats text using pattern-action statements
`sed`	Manipulates text using search-and-replace expressions
`split`	Splits input into several files
`unexpand`	Converts space characters to tab characters
`uniq`	Omits repeated lines
`xargs`	Turns a list of input into arguments that can be used with a command

Table 7-2 Commands that are commonly used within a pipe

```
Where civil blood makes civil hands unclean.
From forth the fatal loins of these two foes
A pair of star-cross'd lovers take their life;
Whole misadventured piteous overthrows
Do with their death bury their parents' strife.
The fearful passage of their death-mark'd love,
And the continuance of their parents' rage,
Which, but their children's end, nought could remove,
Is now the two hours' traffic of our stage;
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

Now suppose you want to replace all lowercase "a" characters with uppercase "A" characters in the preceding file, sort the contents by the first character on each line, double-space the output, and view the results page-by-page. To accomplish these tasks, you can use the following pipe:

```
[root@server1 ~]# cat prologue | tr a A | sort | pr -d | less

2015-08-20 08:06                                                Page 1
A pAir of stAr-cross'd lovers tAke their life;
And the continuAnce of their pArents' rAge,
Do with their deAth bury their pArents' strife.
From Ancient grudge breAk to new mutiny,
From forth the fAtAl loins of these two foes
In fAir VeronA, where we lAy our scene,
Is now the two hours' trAffic of our stAge;
The feArful pAssAge of their deAth-mArk'd love,
The which if you with pAtient eArs Attend,
Two households, both Alike in dignity,
WhAt here shAll miss, our toil shAll strive to mend.
Where civil blood mAkes civil hAnds uncleAn.
Which, but their children's end, nought could remove,
Whole misAdventured piteous overthrows
:
```

The command used in the preceding example displays the final Standard Output to the terminal screen via the `less` command. In many cases, you might want to display the results of the pipe while at the same time save copy in a file on the hard disk. The filter command for this job is the **tee command**, which takes information from Standard Input and sends that information to a file, as well as to Standard Output.

To save a copy of the manipulated prologue before displaying it to the terminal screen with the `less` command, you can use the following command:

```
[root@server1 ~]# cat prologue|tr a A|sort|pr -d|tee newfile|less
2015-08-20 08:06                                                Page 1
A pAir of stAr-cross'd lovers tAke their life;
And the continuAnce of their pArents' rAge,
Do with their deAth bury their pArents' strife.
```

```
From Ancient grudge breAk to new mutiny,
From forth the fAtAl loins of these two foes
In fAir VeronA, where we lAy our scene,
Is now the two hours' trAffic of our stAge;
The feArful pAssAge of their deAth-mArk'd love,
The which if you with pAtient eArs Attend,
Two households, both Alike in dignity,
WhAt here shAll miss, our toil shAll strive to mend.
Where civil blood mAkes civil hAnds uncleAn.
Which, but their children's end, nought could remove,
Whole misAdventured piteous overthrows
:q
[root@server1 ~]# _
[root@server1 ~]# cat newfile
2015-08-20 08:06                                          Page 1
A pAir of stAr-cross'd lovers tAke their life;
And the continuAnce of their pArents' rAge,
Do with their deAth bury their pArents' strife.
From Ancient grudge breAk to new mutiny,
From forth the fAtAl loins of these two foes
In fAir VeronA, where we lAy our scene,
Is now the two hours' trAffic of our stAge;
The feArful pAssAge of their deAth-mArk'd love,
The which if you with pAtient eArs Attend,
Two households, both Alike in dignity,
WhAt here shAll miss, our toil shAll strive to mend.
Where civil blood mAkes civil hAnds uncleAn.
Which, but their children's end, nought could remove,
Whole misAdventured piteous overthrows
[root@server1 ~]# _
```

You can also combine redirection and piping, as long as input redirection occurs at the beginning of the pipe and output redirection occurs at the end of the pipe. An example of this is shown in the following output, which replaces all lowercase a characters with uppercase A characters in the prologue file used in the previous example, then sorts the file, numbers each line, and saves the output to a file called newprologue instead of sending the output to the terminal screen.

```
[root@server1 ~]# tr a A <prologue | sort | nl >newprologue
[root@server1 ~]# cat newprologue
     1  A pAir of stAr-cross'd lovers tAke their life;
     2  And the continuAnce of their pArents' rAge,
     3  Do with their deAth bury their pArents' strife.
     4  From Ancient grudge breAk to new mutiny,
     5  From forth the fAtAl loins of these two foes
     6  In fAir VeronA, where we lAy our scene,
     7  Is now the two hours' trAffic of our stAge;
     8  The feArful pAssAge of their deAth-mArk'd love,
     9  The which if you with pAtient eArs Attend,
```

```
    10 Two households, both Alike in dignity,
    11 WhAt here shAll miss, our toil shAll strive to mend.
    12 Where civil blood mAkes civil hAnds uncleAn.
    13 Which, but their children's end, nought could remove,
    14 Whole misAdventured piteous overthrows
[root@server1 ~]# _
```

Many Linux commands can be used to provide large amounts of useful text information. As a result, Linux administrators often use the `sed` and `awk` filter commands in conjunction with pipes to manipulate text information obtained from these commands.

The `sed` command is typically used to search for a certain string of text, and replaces that text string with another text string using the syntax `s/search/replace/`. For example, the following output demonstrates how `sed` can be used to search for the string "the" and replace it with the string "THE" in the prologue file used earlier:

```
[root@server1 ~]# cat prologue | sed s/the/THE/
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth THE fatal loins of these two foes
A pair of star-cross'd lovers take THEir life;
Whole misadventured piteous overthrows
Do with THEir death bury their parents' strife.
The fearful passage of THEir death-mark'd love,
And THE continuance of their parents' rage,
Which, but THEir children's end, nought could remove,
Is now THE two hours' traffic of our stage;
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

Notice from the preceding output that `sed` only searched for and replaced the first occurrence of the string "the" in each line. To have `sed` globally replace all occurrences of the string "the" in each line, simply append a `g` to the search-and-replace expression:

```
[root@server1 ~]# cat prologue | sed s/the/THE/g
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth THE fatal loins of THEse two foes
A pair of star-cross'd lovers take THEir life;
Whole misadventured piteous overthrows
Do with THEir death bury THEir parents' strife.
The fearful passage of THEir death-mark'd love,
And THE continuance of THEir parents' rage,
Which, but THEir children's end, nought could remove,
Is now THE two hours' traffic of our stage;
The which if you with patient ears attend,
```

```
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

You can also tell sed the specific lines to search by prefixing the search-and-replace expression. For example, to force sed to replace the string "the" with "THE" globally on lines that contain the string "love," you can use the following command:

```
[root@server1 ~]# cat prologue | sed /love/s/the/THE/g
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth the fatal loins of these two foes
A pair of star-cross'd lovers take THEir life;
Whole misadventured piteous overthrows
Do with their death bury their parents' strife.
The fearful passage of THEir death-mark'd love,
And the continuance of their parents' rage,
Which, but their children's end, nought could remove,
Is now the two hours' traffic of our stage;
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

You can also force sed to perform a search-and-replace on certain lines only. To replace the string "the" with "THE" globally on lines 5 to 8 only, you can use the following command:

```
[root@server1 ~]# cat prologue | sed 5,8s/the/THE/g
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
From forth THE fatal loins of THEse two foes
A pair of star-cross'd lovers take THEir life;
Whole misadventured piteous overthrows
Do with THEir death bury THEir parents' strife.
The fearful passage of their death-mark'd love,
And the continuance of their parents' rage,
Which, but their children's end, nought could remove,
Is now the two hours' traffic of our stage;
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

You can also use sed to remove unwanted lines of text. To delete all the lines that contain the word "the," you can use the following command:

```
[root@server1 ~]# cat prologue | sed /the/d
Two households, both alike in dignity,
In fair Verona, where we lay our scene,
From ancient grudge break to new mutiny,
Where civil blood makes civil hands unclean.
```

```
Whole misadventured piteous overthrows
The which if you with patient ears attend,
What here shall miss, our toil shall strive to mend.
[root@server1 ~]# _
```

Like sed, the awk command searches for patterns of text and performs some action on the text found. However, the awk command treats each line of text as a record in a database, and each word in a line as a database field. For example, the line "Hello, how are you?" has four fields: "Hello," "how," "are," and "you?". These fields can be referenced in the awk command using $1, $2, $3, and $4. For example, to display only the first and fourth words only on lines of the prologue file that contains the word "the," you can use the following command:

```
[root@server1 ~]# cat prologue | awk '/the/ {print $1, $4}'
From fatal
A star-cross'd
Do death
The of
And of
Which, children's
Is two
[root@server1 ~]# _
```

By default, the awk command uses space or tab characters as delimiters for each field in a line. Most configuration files on Linux systems, however, are delimited using colon (:) characters. To change the delimiter that awk uses, you can specify the -F option to the command. For example, the following example lists the last 10 lines of the colon-delimited file /etc/passwd and views only the 6th and 7th fields for lines that contain the word "bob" in the last 10 lines of the file:

```
[root@server1 ~]# tail /etc/passwd
news:x:9:13:News server user:/etc/news:/sbin/nologin
smolt:x:490:474:Smolt:/usr/share/smolt:/sbin/nologin
backuppc:x:489:473::/var/lib/BackupPC:/sbin/nologin
pulse:x:488:472:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
gdm:x:42:468::/var/lib/gdm:/sbin/nologin
hsqldb:x:96:96::/var/lib/hsqldb:/sbin/nologin
jetty:x:487:467::/usr/share/jetty:/bin/sh
bozo:x:500:500:bozo the clown:/home/bozo:/bin/bash
bob:x:501:501:Bob Smith:/home/bob:/bin/bash
user1:x:502:502:sample user one:/home/user1:/bin/bash
[root@server1 ~]# tail /etc/passwd | awk -F : '/bob/ {print $6, $7}'
/home/bob /bin/bash
[root@server1 ~]# _
```

Both awk and sed allow you to specify regular expressions in the search pattern.

Shell Variables

A BASH shell has several **variables** in memory at any one time. Recall that a variable is simply a reserved portion of memory containing information that might be accessed. Most variables in the shell are referred to as **environment variables** because they are typically set by the system and contain information that the system and programs access regularly. Users can also create their own custom variables. These variables are called **user-defined variables**. In addition to these two types of variables, special variables are available that are useful when executing commands and creating new files and directories.

Environment Variables

Many environment variables are set by default in the BASH shell. To see a list of these variables and their current values, you can use the **`set` command**, as shown in the following output:

```
[root@server1 ~]# set | less
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:
force_
fignore:histap/bin/bashpend:interactive_comments:login_shell:progcomp:
promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_LINENO=()
BASH_REMATCH=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="2" [2]="45" [3]="1" [4]="release"
[5]="x86_64-redhat-linux-gnu")
BASH_VERSION='4.2.45(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=80
COMP_WORDBREAKS=$' \t\n"\'>< =;|&(:'
DIRSTACK=()
EUID=0
GROUPS=()
HISTCONTROL=ignoredups
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
HOSTNAME=server1
HOSTTYPE=x86_64
ID=0
IFS=$' \t\n'
INCLUDE=
```

```
KDEDIRS=/usr
LANG=en_US.UTF-8
LESSOPEN=‘||/usr/bin/lesspipe.sh %s’
LINES=59
LOGNAME=root
MACHTYPE=x86_64-redhat-linux-gnu
MAIL=/var/spool/mail/root
MAILCHECK=60
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
PIPESTATUS=([0]="0")
PPID=1291
PROMPT_COMMAND=‘printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%%.*}"
"${PWD/#$HOME/~}"‘
PS1=‘[\u@\h \W]\$ ‘
PS2=‘> ‘
PS4=‘+ ‘
PWD=/root
QT_GRAPHICSSYSTEM_CHECKED=1
SELINUX_LEVEL_REQUESTED=
SELINUX_ROLE_REQUESTED=
SELINUX_USE_CURRENT_RANGE=
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
SHLVL=1
SSH_CLIENT=‘192.168.1.100 43480 22’
SSH_CONNECTION=‘192.168.1.100 43480 192.168.1.105 22’
SSH_TTY=/dev/pts/0
TERM=xterm
TMP=/tmp/.colorlsn3e
UID=0
USER=root
XDG_RUNTIME_DIR=/run/user/0
XDG_SESSION_ID=2
:
```

Some environment variables shown in the preceding output are used by programs that require information about the system; the OSTYPE (Operating System TYPE) and SHELL (Pathname to shell) variables are examples from the preceding output. Other variables are used to set the user's working environment; the most common of these include the following:

- *PS1*—The default shell prompt
- *HOME*—The absolute pathname to the user's home directory
- *PWD*—The present working directory in the directory tree
- *PATH*—A list of directories to search for executable programs

The PS1 variable represents the BASH shell prompt. To view the contents of this variable only, you can use the **echo command** and specify the variable name prefixed by the $ shell metacharacter, as shown in the following output:

```
[root@server1 ~]# echo $PS1
[\u@\h \W]\$
[root@server1 ~]# _
```

Note that a special notation is used to define the prompt in the preceding output: \u indicates the user name, \h indicates the host name, and \W indicates the name of the current directory. A list of BASH notation can be found by navigating the manual page for the BASH shell.

To change the value of a variable, you specify the variable name followed immediately by an equal sign (=) and the new value. The following output demonstrates how you can change the value of the PS1 variable. The new prompt takes effect immediately and allows the user to type commands.

```
[root@server1 ~]# PS1="This is the new prompt: #"
This is the new prompt: # _
This is the new prompt: # date
Fri Aug 20 08:16:59 EDT 2015
This is the new prompt: # _
This is the new prompt: # who
(unknown) :0      2015-08-20 07:14 (:0)
root   tty2       2015-08-20 07:15
This is the new prompt: # _
This is the new prompt: # PS1="[\u@\h \W]#"
[root@server1 ~]# _
```

The HOME variable is used by programs that require the pathname to the current user's home directory to store or search for files; therefore, it should not be changed. If the root user logs in to the system, the HOME variable is set to /root; alternatively, the HOME variable is set to /home/user1 if the user named user1 logs in to the system. Recall that the tilde ~ metacharacter represents the current user's home directory; this metacharacter is a pointer to the HOME variable, as shown here:

```
[root@server1 ~]# echo $HOME
/root
[root@server1 ~]# echo ~
/root
[root@server1 ~]# HOME=/etc
[root@server1 root]# echo $HOME
/etc
[root@server1 root]# echo ~
/etc
[root@server1 root]# _
```

Like the HOME variable, the PWD (Print Working Directory) variable is vital to the user's environment and should not be changed. PWD stores the current user's location in the directory tree. It is affected by the cd command and used by other commands such as pwd when

the current directory needs to be identified. The following output demonstrates how this variable works:

```
[root@server1 ~]# pwd
/root
[root@server1 ~]# echo $PWD
/root
[root@server1 ~]# cd /etc
[root@server1 etc]# pwd
/etc
[root@server1 ~]# echo $PWD
/etc
[root@server1 ~]# _
```

The PATH variable is one of the most important variables in the BASH shell, as it allows users to execute commands by typing the command name alone. Recall that most commands are represented by an executable file on the hard drive. These executables are typically stored in directories named bin or sbin in various locations throughout the Linux directory tree. To execute the `ls` command, you could either type the absolute or relative pathname to the file (that is, /usr/bin/ls or ../usr/bin/ls) or simply type the letters "ls" and allow the system to search the directories listed in the PATH variable for a command named `ls`. Sample contents of the PATH variable are shown in the following output:

```
[root@server1 ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@server1 ~]# _
```

In this example, if the user had typed the command `ls` at the command prompt and pressed Enter, the shell would have noticed the lack of a / character in the pathname and proceeded to search for the file ls in the /usr/local/sbin directory, then the /usr/local/bin directory, the /usr/sbin directory, and then the /usr/bin directory before finding the ls executable file. If no ls file is found in any directory in the PATH variable, the shell returns an error message, as shown here with a misspelled command and any similar command suggestions:

```
[root@server1 ~]# lss
bash: lss: command not found...
Similar command is: 'ls'
[root@server1 ~]# _
```

Thus, if a command is located within a directory that is listed in the PATH variable, you can simply type the name of the command on the command line to execute it. The shell will then find the appropriate executable file on the filesystem. All of the commands used in this book so far have been located in directories listed in the PATH variable. However, if the executable file is not in a directory listed in the PATH variable, the user must specify either the absolute or relative pathname to the executable file. The following example uses the myprogram file in the /root directory (a directory that is not listed in the PATH variable):

```
[root@server1 ~]# pwd
/root
[root@server1 ~]# ls -F
Desktop/ myprogram*
[root@server1 ~]# myprogram
bash: myprogram: command not found...
```

```
[root@server1 ~]# /root/myprogram
This is a sample program.
[root@server1 ~]# ./myprogram
This is a sample program.
[root@server1 ~]# cp myprogram /usr/bin
[root@server1 ~]# myprogram
This is a sample program.
[root@server1 ~]# _
```

After the myprogram executable file was copied to the /usr/bin directory in the preceding output, the user was able to execute it by simply typing its name, because the /usr/bin directory is listed in the PATH variable. Table 7-3 provides a list of environment variables used in most BASH shells.

Variable	Description
BASH	The full path to the BASH shell
BASH_VERSION	The version of the current BASH shell
DISPLAY	The variable used to redirect the output of X Windows to another computer or device that allows remote X connections using the `xhost` command
ENV	The location of the BASH run-time configuration file (usually ~/.bashrc)
EUID	The effective UID (User ID) of the current user
HISTFILE	The filename used to store previously entered commands in the BASH shell (usually ~/.bash_history)
HISTFILESIZE	The number of previously entered commands that can be stored in the HISTFILE upon logout for use during the next login; it is typically 1000 commands
HISTSIZE	The number of previously entered commands that will be stored in memory during the current login session; it is typically 1000 commands
HOME	The absolute pathname of the current user's home directory
HOSTNAME	The host name of the Linux system
LC_ALL	Defines the locale used by the iconv function within the shell for character encoding (e.g. Unicode UTF-8, ISO-8859), overriding the default locale stored in /usr/bin/locale
LOGNAME	The user name of the current user used when logging in to the shell
MAIL	The location of the mailbox file (where e-mail is stored)
OSTYPE	The current operating system
PATH	The directories to search for executable program files in the absence of an absolute or relative pathname containing a / character
PS1	The current shell prompt
PWD	The current working directory
RANDOM	The variable that creates a random number when accessed
SHELL	The absolute pathname of the current shell
TERM	The variable used to determine the terminal settings; it is typically set to "linux" or "xterm" on newer Linux systems and "console" on older Linux systems
TERMCAP	The variable used to determine the terminal settings on Linux systems that use a TERMCAP database (/etc/termcap)

Table 7-3 Common BASH environment variables

User-Defined Variables

You can set your own variables using the same method discussed earlier to change the contents of existing environment variables. To do so, you simply specify the name of the variable (known as the **variable identifier**) immediately followed by the equal sign (=) and the new contents. When creating new variables, it is important to note the following features of variable identifiers:

- They can contain alphanumeric characters (0–9, A–Z, a–z), the dash (−) character, or the underscore (_) character.
- They must not start with a number.
- They are typically capitalized to follow convention (e.g., HOME, PATH).

To create a variable called MYVAR with the contents "This is a sample variable" and display its contents, you can use the following commands:

```
[root@server1 ~]# MYVAR="This is a sample variable"
[root@server1 ~]# echo $MYVAR
This is a sample variable
[root@server1 ~]# _
```


The preceding command created a variable that is available to the current shell. Most commands that are run by the shell are run in a separate **subshell**, which is created by the current shell. Any variables created in the current shell are not available to those subshells and the commands running within them. Thus, if a user creates a variable to be used within a certain program such as a database editor, that variable should be exported to all subshells using the **`export` command** to ensure that all programs started by the current shell have the ability to access the variable.

As explained earlier in this chapter, all environment variables in the BASH shell can be listed using the `set` command; user-defined variables are also indicated in this list. Similarly, to see a list of all exported environment and user-defined variables in the shell, you can use the **`env` command**. Because the outputs of `set` and `env` are typically large, you would commonly redirect the Standard Output of these commands to the `grep` command to display certain lines only.

To see the difference between the `set` and `env` commands as well as export the MYVAR variable created earlier, you can perform the following commands:

```
[root@server1 ~]# set | grep MYVAR
MYVAR='This is a sample variable.'
[root@server1 ~]# env | grep MYVAR
[root@server1 ~]# _
[root@server1 ~]# export MYVAR
[root@server1 ~]# env | grep MYVAR
MYVAR=This is a sample variable.
[root@server1 ~]# _
```

Not all environment variables are exported; the PS1 variable is an example of a variable that does not need to be available to subshells and is not exported as a result. However, it is good form to export user-defined variables because they will likely be used by processes that run in

subshells. This means, to create and export a user-defined variable called MYVAR2, you can use the `export` command alone, as shown in the following output:

```
[root@server1 ~]# export MYVAR2="This is another sample variable"
[root@server1 ~]# set | grep MYVAR2
MYVAR2='This is another sample variable.'
_=MYVAR2
[root@server1 ~]# env | grep MYVAR2
MYVAR2=This is another sample variable.
[root@server1 ~]# _
```

You can also remove a variable from memory using the `unset variablename` command.

Other Variables

Other variables are not displayed by the `set` or `env` commands; these variables perform specialized functions in the shell.

The UMASK variable used earlier in this textbook is an example of a special variable that performs a special function in the BASH shell and must be set by the `umask` command. Also recall that when you type the `cp` command, you are actually running an alias to the `cp -i` command. Aliases are shortcuts to commands stored in special variables that can be created and viewed using the **`alias` command**. To create an alias to the command `mount -t ext2 /dev/fd0 /mnt/floppy` called mf and view it, you can use the following commands:

```
[root@server1 ~]# alias mf="mount -t ext2 /dev/fd0 /mnt/floppy"
[root@server1 ~]# alias
alias cp='cp -i'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d.* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mf='mount -t ext2 /dev/fd0 /mnt/floppy'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
[root@server1 ~]# _
```

Now, you simply need to run the `mf` command to mount a floppy device that contains an ext2 filesystem to the /mnt/floppy directory, as shown in the following output:

```
[root@server1 ~]# mf
[root@server1 ~]# mount | grep fd0
/dev/fd0 on /mnt/floppy type ext2 (rw)
[root@server1 ~]# _
```

You can also create aliases to multiple commands, provided they are separated by the ; metacharacter introduced in Chapter 2. To create and test an alias called dw that runs the `date` command followed by the `who` command, you can do the following:

```
[root@server1 ~]# alias dw="date;who"
[root@server1 ~]# alias
alias cp='cp -i'
alias dw='date;who'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d.* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mf='mount -t ext2 /dev/fd0 /mnt/floppy'
alias mv='mv -i'
alias rm='rm -i'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
[root@server1 ~]# dw
Thu Oct 2 08:07:22 EDT 2015
(unknown) :0          2015-10-02 07:14 (:0)
root       tty2       2015-10-02 07:15
[root@server1 ~]# _
```

It is important to use unique alias names because the shell searches for them before it searches for executable files. If you create an alias called who, that alias would be used instead of the `who` command on the filesystem.

Environment Files

Recall that variables are stored in memory. When a user exits the BASH shell, all variables stored in memory are destroyed along with the shell itself. To ensure that variables are accessible to a shell at all times, you must place variables in a file that is executed each time a user logs in and starts a BASH shell. These files are called **environment files**. Common BASH shell environment files and the order in which they are typically executed are as follows:

/etc/profile

/etc/bashrc

~/.bashrc

~/.bash_profile

~/.bash_login

~/.profile

The BASH runtime configuration files (/etc/bashrc and ~/.bashrc) are typically used to set aliases and variables that must be present in the BASH shell. They are executed immediately after a new login as well as when a new BASH shell is created after login. The /etc/bashrc file contains aliases and variables for all users on the system, whereas the ~/.bashrc file contains aliases and variables for a specific user.

The other environment files are only executed after a new login. The /etc/profile file is executed after login for all users on the system and sets most environment variables, such as HOME and PATH. After /etc/profile finishes executing, the home directory of the user is searched for the hidden environment files. bash_profile, .bash_login, and .profile. If these files exist, the first one found is executed; as a result, only one of these files is typically used. These hidden environment files allow a user to set customized variables independent of BASH shells used by other users on the system; any values assigned to variables in these files override those set in /etc/profile, /etc/bashrc, and ~/.bashrc due to the order of execution.

To add a variable to any of these files, you simply add a line that has the same format as the command used on the command line. To add the MYVAR2 variable used previously to the .bash_profile file, simply edit the file using a text editor such as vi and add the line `export MYVAR2="This is another sample variable"` to the file.

Variables are not the only type of information that can be entered into an environment file; any command that can be executed on the command line can also be placed inside any environment file. If you want to set the UMASK to 077, display the date after each login, and create an alias, you can add the following lines to one of the hidden environment files in your home directory:

```
umask 077
date
alias dw="date;who"
```

Also, you might want to execute cleanup tasks upon exiting the shell; to do this, simply add those cleanup commands to the .bash_logout file in your home directory.

Shell Scripts

In the previous section, you learned that the BASH shell can execute commands that exist within environment files. The BASH shell also has the ability to execute other text files containing commands and special constructs. These files are referred to as **shell scripts** and are typically used to create custom programs that perform administrative tasks on Linux systems. Any command that can be entered on the command line in Linux can be entered into a shell script because it is a BASH shell that interprets the contents of the shell script itself. The most basic shell script is one that contains a list of commands, one per line, for the shell to execute in order, as shown here in the text file called myscript:

```
[root@server1 ~]# cat myscript
#!/bin/bash
#this is a comment
date
who
ls -F /
[root@server1 ~]# _
```

The first line in the preceding shell script (`#!/bin/bash`) is called a **hashpling**; it specifies the pathname to the shell that interprets the contents of the shell script. Different shells can use different constructs in their shell scripts. Thus, it is important to identify which shell was used to create a particular shell script. The hashpling allows a user who uses the C shell to use a BASH shell when executing the myscript shell script shown previously. The second line of the shell script is referred to as a comment because it begins with a # character and is ignored by the shell; the only exception to this is the hashpling on the first line of a shell script. The remainder of the shell script shown in the preceding output consists of three commands that will be executed by the shell in order: `date`, `who`, and `ls`.

If you have read permission to a shell script, you can execute the shell script by starting another BASH shell and specifying the shell script as an argument. To execute the myscript shell script shown earlier, you can use the following command:

```
[root@server1 ~]# bash myscript
Fri Aug 20 11:36:18 EDT 2015
user1     tty1       2015-08-20 07:47 (:0)
root      tty2       2015-08-20 11:36
bin/      dev/       home/         media/  proc/    sbin/     sys/  var/
boot/     etc/       lib/          mnt/    public/  selinux/  tmp/
data/     extras/    lost+found/   opt/    root/    srv/      usr/
[root@server1 ~]# _
```

Alternatively, if you have read and execute permission to a shell script, you can execute the shell script like any other executable program on the system, as shown here using the myscript shell script:

```
[root@server1 ~]# chmod a+x myscript
[root@server1 ~]# ./myscript
Fri Aug 20 11:36:58 EDT 2015
user1     tty1       2015-08-20 07:47 (:0)
root      tty2       2015-08-20 11:36
bin/      dev/       home/         media/  proc/    sbin/     sys/  var/
boot/     etc/       lib/          mnt/    public/  selinux/  tmp/
data/     extras/    lost+found/   opt/    root/    srv/      usr/
[root@server1 ~]# _
```

The preceding output is difficult to read because the output from each command is not separated by blank lines or identified by a label. Utilizing the `echo` command results in a more user-friendly myscript, as shown here:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo "Today's date is:"
date
echo ""
echo "The people logged into the system include:"
who
echo ""
echo "The contents of the / directory are:"
```

```
ls -F /
[root@server1 ~]#./ myscript
Today's date is:
Fri Aug 20 11:37:24 EDT 2015

The people logged into the system include:
user1   tty1      2015-08-20 07:47 (:0)
root    tty2      2015-08-20 11:36
The contents of the / directory are:
bin/   dev/     home/       media/  proc/    sbin/     sys/  var/
boot/  etc/     lib/        mnt/    public/  selinux/  tmp/
data/  extras/  lost+found/ opt/    root/    srv/      usr/
[root@server1 ~]# _
```

Escape Sequences

In the previous example, you used the `echo` command to manipulate data that appeared on the screen. The `echo` command also supports several special notations called **escape sequences**. You can use escape sequences to further manipulate the way text is displayed to the terminal screen, provided the `-e` option is specified to the `echo` command. Table 7-4 provides a list of these echo escape sequences.

The escape sequences listed in Table 7-4 can be used to further manipulate the output of the myscript shell script used earlier, as shown in the following example:

```
[root@server1 ~]# cat myscript
#!/bin/bash
```

Escape Sequence	Description
\???	Inserts an ASCII character represented by a three-digit octal number (???)
\\	Backslash
\a	ASCII beep
\b	Backspace
\c	Prevents a new line following the command
\f	Form feed
\n	Starts a new line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Table 7-4 Common echo escape sequences

```
echo -e "Today's date is: \c"
date
echo –e "\nThe people logged into the system include:"
who
echo –e "\nThe contents of the / directory are:"
ls –F /
[root@server1 ~]#./ myscript
Today's date is: Fri Aug 20 11:44:24 EDT 2015
The people logged into the system include:
user1     tty1       2015-08-20 07:47 (:0)
root      tty2       2015-08-20 11:36

The contents of the / directory are:
bin/      dev/       home/         media/  proc/    sbin/     sys/  var/
boot/     etc/       lib/          mnt/    public/  selinux/  tmp/
data/     extras/    lost+found/   opt/    root/    srv/      usr/
[root@server1 ~]# _
```

Notice from preceding output that the \c escape sequence prevented the newline character at the end of the output "Today's date is:" when myscript was executed. Similarly, newline characters (\n) were inserted prior to displaying "The people logged into the system include:" and "The contents of the / directory are:" to create blank lines between command outputs. This eliminated the need for using the echo "" command shown earlier.

Reading Standard Input

At times, a shell script might need input from the user executing the program; this input can then be stored in a variable for later use. The **read command** takes user input from Standard Input and places it in a variable specified by an argument to the read command. After the input has been read into a variable, the contents of that variable can then be used, as shown in the following shell script:

```
[root@server1 ~]# cat newscript
#!/bin/bash
echo –e "What is your name? --> \c"
read USERNAME
echo "Hello $USERNAME"
[root@server1 ~]# chmod a+x newscript
[root@server1 ~]# ./newscript
What is your name? -->Fred
Hello Fred
[root@server1 ~]# _
```

Note from the preceding output that the echo command used to pose a question to the user ends with --> to simulate an arrow prompt on the screen and the \c escape sequence to place the cursor after the arrow prompt; this is common among Linux administrators when writing shell scripts.

Decision Constructs

Decision constructs are the most common type of construct used in shell scripts. They alter the flow of a program based on whether a command in the program completed successfully or based on a decision that the user makes in response to a question posed by the program. Figures 7-4 and 7-5 illustrate some decision constructs.

The `if` Construct

The most common type of decision construct, the `if` construct, has the following syntax:

```
if this is true
then
do these commands
elif this is true
then
do these commands
else
do these commands
fi
```

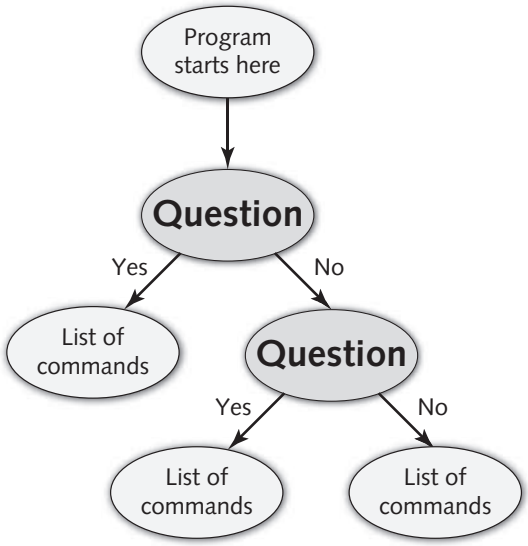

Figure 7-4 A two-question decision construct

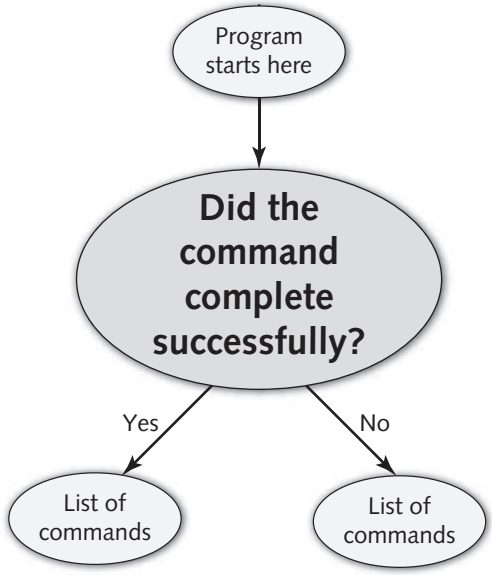

Figure 7-5 A command-based decision construct

Some common rules govern `if` constructs:

1. elif (else if) and else statements are optional.
2. You can have an unlimited number of elif statements.
3. The *do these* commands section can consist of multiple commands, one per line.
4. The *do these* commands section is typically indented from the left-hand side of the text file for readability but does not need to be.
5. The end of the statement must be a backward "if" (`fi`).
6. The *this is true* part of the if syntax shown earlier can be a command or a **test statement**:
   - Commands return true if they perform their function properly.
   - Test statements are enclosed within square brackets [ ] or prefixed by the word "test" and used to test certain conditions on the system.

In the following example, a basic `if` construct is used to ensure that the /etc/hosts file is only copied to the /etc/sample directory if that directory could be created successfully:


```
[root@server1 ~]# cat testmkdir
#!/bin/bash
if mkdir /etc/sample
then
cp /etc/hosts /etc/sample
echo "The hosts file was successfully copied to /etc/sample"
else
echo "The /etc/sample directory could not be created."
fi
[root@server1 ~]# chmod a+x testmkdir
[root@server1 ~]# ./testmkdir
The hosts file was successfully copied to /etc/sample
[root@server1 ~]# _
```

In the preceding output, the `mkdir /etc/sample` command is always run. If it runs successfully, the shell script proceeds to the `cp /etc/hosts /etc/sample` and `echo "The hosts file was successfully copied to /etc/sample"` commands. If the `mkdir /etc/sample` command is unsuccessful, the shell script skips ahead and executes the `echo "The /etc/sample directory could not be created."` command. If there were more lines of text following the `fi` in the preceding shell script, they are executed after the `if` construct, regardless of its outcome.

Often, it is useful to use the `if` construct to alter the flow of the program given input from the user. Recall the myscript shell script used earlier:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo -e "Today's date is: \c"
date
echo –e "\nThe people logged into the system include:"
who
echo –e "\nThe contents of the / directory are:"
```

```
ls -F /
[root@server1 ~]# _
```

To ask the user whether to display the contents of the / directory, you could use the following `if` construct in the myscript file:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo -e "Today's date is: \c"
date
echo -e "\nThe people logged into the system include:"
who
echo -e "\nWould you like to see the contents of /?(y/n)--> \c"
read ANSWER
if [ $ANSWER = "y" ]
then
echo -e "\nThe contents of the / directory are:"
ls -F /
fi
[root@server1 ~]#./myscript
Today's date is: Fri Aug 20 11:47:14 EDT 2015

The people logged into the system include:
user1   tty1      2015-08-20 07:47 (:0)
root    tty2      2015-08-20 11:36
Would you like to see the contents of /?(y/n)--> y

The contents of the / directory are:
bin/     dev/      home/         media/  proc/    sbin/     sys/  var/
boot/    etc/      lib/          mnt/    public/  selinux/  tmp/
data/    extras/   lost+found/   opt/    root/    srv/      usr/
[root@server1 ~]# _
```

In the preceding output, the test statement `[ $ANSWER = "y" ]` is used to test whether the contents of the ANSWER variable are equal to the letter "y." Any other character in this variable causes this test statement to return false, and the directory listing is then skipped altogether. The type of comparison used previously is called a string comparison because two values are compared for strings of characters; it is indicated by the operator of the test statement, which is the equal sign (=) in this example. Table 7-5 shows a list of common operators used in test statements and their definitions.

The test statement `[ $ANSWER = "y" ]` is equivalent to the test statement `test $ANSWER = "y"`.

It is important to include a space character after the beginning square bracket and before the ending square bracket; otherwise, the test statement produces an error.

Test Statement	Returns True if:
`[ A = B ]`	String A is equal to String B.
`[ A != B ]`	String A is not equal to String B.
`[ A -eq B ]`	A is numerically equal to B.
`[ A -ne B ]`	A is numerically not equal to B.
`[ A –lt B ]`	A is numerically less than B.
`[ A –gt B ]`	A is numerically greater than B.
`[ A –le B ]`	A is numerically less than or equal to B.
`[ A –ge B ]`	A is numerically greater than or equal to B.
`[ -r A ]`	A is a file/directory that exists and is readable (r permission).
`[ -w A ]`	A is a file/directory that exists and is writable (w permission).
`[ -x A ]`	A is a file/directory that exists and is executable (x permission).
`[ -f A ]`	A is a file that exists.
`[ -d A ]`	A is a directory that exists.

Table 7-5 Common test statements

Test Statement	Returns True if:
`[ A = B –o C = D ]`	String A is equal to String B OR String C is equal to String D.
`[ A = B –a C = D ]`	String A is equal to String B AND String C is equal to String D.
`[ ! A = B ]`	String A is NOT equal to String B.

Table 7-6 Special operators in test statements

You can combine any test statement with another test statement using the comparison operators `–o` (OR) and `-a` (AND). To reverse the meaning of a test statement, you can use the `!` (NOT) operator. Table 7-6 provides some examples of using these operators in test statements.

One test statement can contain several `–o`, `-a`, and `!` operators.

By modifying the myscript shell script in the previous output, you can proceed with the directory listing if the user enters "y" or "Y," as shown in the following example:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo -e "Today's date is: \c"
date
```

```
echo –e "\nThe people logged into the system include:"
who
echo –e "\nWould you like to see the contents of /?(y/n) --> \c"
read ANSWER
if [ $ANSWER = "y" –o $ANSWER = "Y" ]
then
echo –e "\nThe contents of the / directory are:"
ls –F /
fi
[root@server1 ~]#./ myscript
Today's date is: Fri Aug 20 12:01:22 EDT 2015

The people logged into the system include:
user1   tty1      2015-08-20 07:47 (:0)
root    tty2      2015-08-20 11:36

Would you like to see the contents of /?(y/n)--> y

The contents of the / directory are:
bin/   dev/    home/        media/  proc/    sbin/     sys/   var/
boot/  etc/    lib/         mnt/    public/  selinux/  tmp/
data/  extras/ lost+found/  opt/    root/    srv/      usr/
[root@server1 ~]# _
```

The `case` Construct The `if` construct used earlier is well suited for a limited number of choices. In the following example, which uses the myscript example presented earlier, several `elif` statements perform tasks based on user input:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo –e "What would you like to see?
Todays date (d)
Currently logged in users (u)
The contents of the / directory (r)

Enter your choice(d/u/r)--> \c"
read ANSWER
if [ $ANSWER = "d" –o $ANSWER = "D" ]
then
echo -e "Today's date is: \c"
date
elif [ $ANSWER = "u" –o $ANSWER = "U" ]
then
echo –e "\nThe people logged into the system include:"
who
elif [ $ANSWER = "r" –o $ANSWER = "R" ]
then
echo –e "\nThe contents of the / directory are:"
ls –F /
```

```
fi
[root@server1 ~]# _
[root@server1 ~]#./ myscript
What would you like to see?
Todays date (d)
Currently logged in users (u)
The contents of the / directory (r)

Enter your choice(d/u/r)--> d
Today's date is: Fri Aug 20 12:13:12 EDT 2015
[root@server1 ~]# _
```

The preceding shell script becomes increasingly difficult to read as the number of choices available increases. Thus, when presenting several choices, it is commonplace to use a `case` construct. The syntax of the `case` construct is as follows:

```
case variable in
pattern1 )  do this
             ;;
pattern2 )  do this
             ;;
pattern3 )  do this
            ;;
esac
```

The `case` statement compares the value of a variable with several different patterns of text or numbers. When a match occurs, the commands to the right of the pattern are executed (*do this* in the preceding syntax). As with the `if` construct, the `case` construct must be ended by a backward "case" (`esac`).

An example that simplifies the previous myscript example by using the `case` construct is shown in the following output:

```
[root@server1 ~]# cat myscript
#!/bin/bash
echo –e "What would you like to see?
Todays date (d)
Currently logged in users (u)
The contents of the / directory (r)

Enter your choice(d/u/r)--> \c"
read ANSWER

case $ANSWER in
  d | D ) echo -e "\nToday's date is: \c"
        date
        ;;
  u | U ) echo –e "\nThe people logged into the system include:"
        who
        ;;
```

```
  r | R ) echo -e "\nThe contents of the / directory are:"
        ls -F /
        ;;
     *) echo -e "Invalid choice! \a"
        ;;
esac
[root@server1 ~]#./ myscript
What would you like to see?
Todays date (d)
Currently logged in users (u)
The contents of the / directory (r)

Enter your choice(d/u/r)--> d
Today's date is: Fri Aug 20 12:33:08 EDT 2015
[root@server1 ~]# _
```

The preceding example prompts the user with a menu and allows the user to select an item that is then placed into the ANSWER variable. If the ANSWER variable is equal to the letter "d" or "D," the `date` command is executed; however, if the ANSWER variable is equal to the letter "u" or "U," the `who` command is executed, and if the ANSWER variable is equal to the letter "r" or "R," the `ls` command is executed. If the ANSWER variable contains something other than the aforementioned letters, the * wildcard metacharacter matches it and prints an error message to the screen. As with `if` constructs, any statements present in the shell script following the `case` construct are executed after the `case` construct.

The && and || Constructs Although the `if` and `case` constructs are versatile, when only one decision needs to be made during the execution of a program, it's faster to use the `&&` and `||` constructs. The syntax of these constructs is listed as follows:

```
command  &&  command
command  ||  command
```

For the preceding `&&` syntax, the command on the right of the `&&` construct is executed only if the command on the left of the `&&` construct completed successfully. The opposite is true for the `||` syntax; the command on the right of the `||` construct is executed only if the command on the left of the `||` construct did not complete successfully.

Consider the testmkdir example presented earlier in this chapter:

```
[root@server1 ~]# cat testmkdir
#!/bin/bash
if mkdir /etc/sample
then
cp /etc/hosts /etc/sample
echo "The hosts file was successfully copied to /etc/sample"
else
echo "The /etc/sample directory could not be created."
fi
[root@server1 ~]# _
```

You can rewrite the preceding shell script utilizing the && construct as follows:

```
[root@server1 ~]# cat testmkdir
#!/bin/bash
mkdir /etc/sample && cp /etc/hosts /etc/sample
[root@server1 ~]# _
```

The preceding shell script creates the directory /etc/sample and only copies the /etc/hosts file to it if the `mkdir /etc/sample` command was successful. You can instead use the || construct to generate error messages if one of the commands fails to execute properly:

```
[root@server1 ~]# cat testmkdir
#!/bin/bash
mkdir /etc/sample || echo "Could not create /etc/sample"
cp /etc/hosts /etc/sample || echo "Could not copy /etc/hosts"
[root@server1 ~]#_
```

Loop Constructs

To execute commands repetitively, you can write shell scripts that contain loop constructs. Like decision constructs, **loop constructs** alter the flow of a program based on the result of a particular statement. But unlike decision constructs, which run different parts of a program depending on the results of the test statement, a loop construct simply repeats the entire program. Although several loop constructs are available within the BASH shell, the two most common are `for` and `while`.

The `for` Construct The `for` construct is the most useful looping construct for Linux administrators because it can be used to process a list of objects, such as files, directories, users, printers, and so on. The syntax of the `for` construct is as follows:

```
for var_name in string1 string2 string3 … …
do
these commands
done
```

When a `for` construct is executed, it creates a variable (`var_name`), sets its value equal to `string1`, and executes the commands between `do` and `done`, which can access the `var_name` variable. Next, the `for` construct sets the value of `var_name` to `string2` and executes the commands between `do` and `done` again. Following this, the `for` construct sets the value of `var_name` to `string3` and executes the commands between `do` and `done` again. This process repeats as long as there are strings to process. Thus, if there are three strings, the `for` construct will execute three times. If there are 20 strings, the `for` construct will execute 20 times.

The following example uses the `for` construct to e-mail a list of users with a new schedule:

```
[root@server1 ~]# cat emailusers
#!/bin/bash
for NAME in bob sue mary jane frank lisa jason
do
mail -s "Your new project schedule" < newschedule $NAME
```

```
echo "$NAME was emailed successfully"
done
[root@server1 ~]# _
[root@server1 ~]# chmod a+x emailusers
[root@server1 ~]# ./emailusers
bob was emailed successfully
sue was emailed successfully
mary was emailed successfully
jane was emailed successfully
frank was emailed successfully
lisa was emailed successfully
jason was emailed successfully
[root@server1 ~]# _
```

When the `for` construct in the preceding example is executed, it creates a NAME variable and sets its value to bob. Then it executes the `mail` command to email bob the contents of the newschedule file with a subject line of `Your new project schedule`. Next, it sets the NAME variable to sue and executes the `mail` command to send sue the same email. This process is repeated until the last person receives the email.

A more common use of the `for` construct within shell scripts is to process several files. The following example renames each file within a specified directory to include a `.txt` extension.

```
[root@server1 ~]# ls stuff
file1 file2 file3 file4 file5 file6 file7 file8
[root@server1 ~]# _
[root@server1 ~]# cat multiplerename
#!/bin/bash
echo –e "What directory has the files that you would like to rename?
  --> \c"
read DIR
for NAME in $DIR/*
do
mv $NAME $NAME.txt
done
[root@server1 ~]# _
[root@server1 ~]# chmod a+x multiplerename
[root@server1 ~]# ./multiplerename
What directory has the files that you would like to rename? --> stuff
[root@server1 ~]# ls stuff
file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt file7.txt
file8.txt
[root@server1 ~]# _
```

When the `for` construct in the previous example is executed, it sets the list of strings to stuff/* (which expands to file1 file2 file3 file4 file5 file6 file7 file8). It then creates a NAME variable, sets its value to file1, and executes the `mv` command to rename file1 to file1.txt. Next, the `for` construct sets the value of the NAME variable to file2 and executes the `mv` command to rename file2 to file2.txt. This is repeated until all of the files have been processed. Note that you can also use the `seq` command to generate a list of numbers for use within a `for` loop.

The `while` Construct The `while` construct is another common loop construct used within shell scripts. Unlike the `for` construct, the `while` construct begins with a test statement. As long as (or while) the test statement returns true, the commands within the loop construct are executed. When the test statement returns false, the commands within the `while` construct stop executing. A `while` construct typically contains a variable, called a **counter variable**, whose value changes each time through the loop. For the `while` construct to work properly, it must be set up so that, when the counter variable reaches a certain value, the test statement returns false. This prevents the loop from executing indefinitely.

The syntax of the `while` construct is as follows:

```
while this returns true
do
these commands
done
```

The following example illustrates the general use of the `while` construct.

```
[root@server1 ~]# cat echorepeat
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 7 ]
do
echo "All work and no play makes Jack a dull boy" >> /tmp/redrum
COUNTER=`expr $COUNTER + 1`
done
[root@server1 ~]# _
[root@server1 ~]# chmod a+x echorepeat
[root@server1 ~]# ./echorepeat
[root@server1 ~]# cat /tmp/redrum
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
All work and no play makes Jack a dull boy
[root@server1 ~]# _
```

The echorepeat shell script shown above creates a counter variable called COUNTER and sets its value to 0. Next, the `while` construct uses a test statement to determine whether the value of the COUNTER variable is less than 7 before executing the commands within the loop. Since the initial value of COUNTER variable is 0, it appends the text `All work and no play makes Jack a dull boy` to the /tmp/redrum file and increments the value of the COUNTER variable to 1. Note the backquotes surrounding the `expr` command, which are required to numerically add 1 to the COUNTER variable (0 + 1 = 1). Because the value of the COUNTER variable at this stage (1) is still less than 7, the `while` construct executes the commands again. This process repeats until the value of the COUNTER variable is equal to 8.

You can use `true` or `:` in place of a test statement to create a `while` construct that executes indefinitely.

Chapter Summary

- Three components are available to commands: Standard Input (stdin), Standard Output (stdout), and Standard Error (stderr). Not all commands use every component.
- Standard Input is typically user input taken from the keyboard, whereas Standard Output and Standard Error are sent to the terminal screen by default.
- You can redirect the Standard Output and Standard Error of a command to a file using redirection symbols. Similarly, you can use redirection symbols to redirect a file to the Standard Input of a command.
- To redirect the Standard Output from one command to the Standard Input of another, you must use the pipe symbol (|).
- Most variables available to the BASH shell are environment variables that are loaded into memory after login from environment files.
- You can create your own variables in the BASH shell and export them to programs started by the shell. These variables can also be placed in environment files, so that they are loaded into memory on every shell login.
- The UMASK variable and command aliases are special variables that must be set using a certain command.
- Shell scripts can be used to execute several Linux commands.
- Decision constructs can be used within shell scripts to execute certain Linux commands based on user input or the results of a certain command.
- Loop constructs can be used within shell scripts to execute a series of commands repetitively.

Key Terms

| A shell metacharacter used to pipe Standard Output from one command to the Standard Input of another command.

< A shell metacharacter used to obtain Standard Input from a file.

> A shell metacharacter used to redirect Standard Output and Standard Error to a file.

`alias` command A command used to create special variables that are shortcuts to longer command strings.

counter variable A variable that is altered by loop constructs to ensure that commands are not executed indefinitely.

decision construct A special construct used in a shell script to alter the flow of the program based on the outcome of a command or contents of a variable. Common decision constructs include `if`, `case`, `&&`, and `||`.

`echo` command A command used to display or echo output to the terminal screen. It might utilize escape sequences.