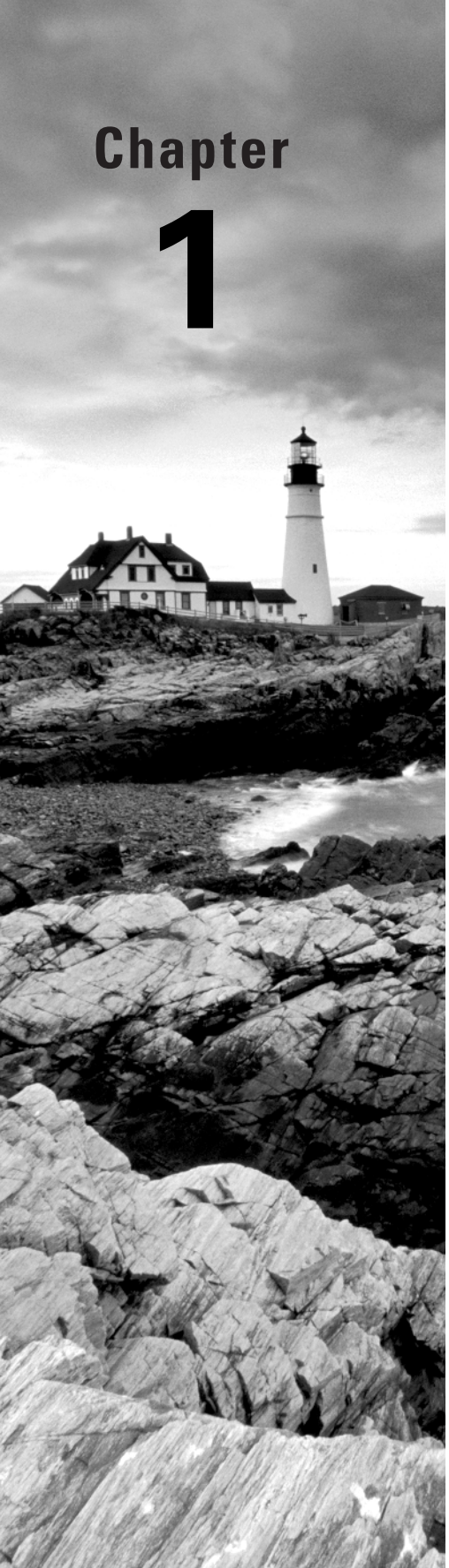
Exam 101-500

PART I



Chapter

1

Exploring Linux Command-Line Tools

OBJECTIVES

- ✓ 103.1 Work on the command line
- ✓ 103.2 Process text streams using filters
- ✓ 103.4 Use streams, pipes, and redirects
- ✓ 103.7 Search text files using regular expressions
- ✓ 103.8 Basic file editing

In the original Linux years, to get anything done you had to work with the *Gnu/Linux shell*. The shell is a special interactive utility that allows users to run programs, manage files, supervise processes, and so on. The shell provides a prompt at which you can enter text-based commands. These commands are actually programs. Although there are literally thousands of these programs, in this chapter we'll be focusing on a few basic commands as well as fundamental shell concepts.

Understanding Command-Line Basics

While it is highly likely that you have had multiple exposures to many of the commands in this chapter, you may not know all of them, and there may be some shell commands you are using in an ineffective manner. In addition, you may have incorrect ideas concerning distributions. Thus, we'll start with the basics, such as distribution differences, how to reach a shell, the various shell options available, how to use a shell, and so on.

Discussing Distributions

Before we look at shells, an important topic to discuss is distributions (also called *distros*). Although it is tempting to think that Linux distributions are all the same and only a few differences exist between them, that is a fallacy. Think of the Linux kernel as a car's engine and a distribution as the car's features. Between manufacturers and models, car features are often different. If you use a rented car, you have to take a few minutes to adjust the seat, view the various car controls, and figure out how to use them prior to driving it. This is also true with different distributions. While they all have the Linux kernel (car engine) at their core, their various features are different, and that can include differences at the command line.

If would like to follow along and try out the various commands in this book, it is helpful to know which distros to use. Because the LPIC-1 V5.0 certification exam is not going to change after its release, it is best to use a selection of Linux distributions that were available during the exam's development. It is incorrect to think that using a distribution's latest version is better. Instead, it is fine to use the same distributions we did while writing the book, which were the CentOS 7 Everything, Ubuntu Desktop 18-04 LTS, Fedora 29 Workstation, and openSUSE 15 Leap distros.

Reaching a Shell

After you install your Linux system or virtual environment distro, set it up, and boot it, you can typically reach a command-line terminal by pressing the Ctrl+Alt+F2 key combination (which gets you to the tty2 terminal), and log in using a standard user account (one without super user privileges). Typically, you create a standard user account when installing a Linux distribution.

If you want to use your Linux distribution's graphical user interface (GUI), you can log in and then open a terminal emulator to reach the command line via the following:

- On an Ubuntu Workstation distro, press Ctrl+Alt+T.
- On a CentOS 7 Everything and a Fedora 29 Workstation distro, click the Activities menu option, enter **`term`** in the search bar, and select the resulting terminal icon.
- On an openSUSE 15 Leap distro, click the Application Menu icon on the screen's bottom left side, enter **`term`** in the search bar, and select one of the resulting terminal icons.

Exploring Your Linux Shell Options

When you successfully log into a tty terminal (such as tty2) or open a GUI terminal emulator program to reach a command-line prompt, the program providing that prompt is a shell. While the Bash shell program is the most popular and commonly used by the various Linux distributions, there are a few others you need to know:

Bash The GNU Bourne Again shell (Bash), first released in 1989, is commonly used as the default shell for Linux user accounts. The Bash shell was developed by the GNU project as a replacement for the standard Unix operating system shell, called the Bourne shell (named for its creator). It is also available for Windows 10, macOS, and Solaris operating systems.

Dash The Debian Almquist shell (Dash) was originally released in 2002. This smaller shell does not allow command-line editing or command history (covered later in this chapter), but it does provide faster shell program (also called a *script*) execution.

KornShell The KornShell was initially released in 1983 but was proprietary software until 2000. It was invented by David Korn of Bell Labs. It is a programming shell compatible with the Bourne shell but supports advanced programming features, such as those available in the C programming languages.

tcsh Originally released in 1981, the TENEX C shell is an upgraded version of the C Shell. It added command completion, which was a nice feature in the TENEX operating system. In addition, tcsh incorporates elements from the C programming language into shell scripts.

Z shell The Z shell was first released in 1990. This advanced shell incorporates features from Bash, tcsh, and KornShell. Advanced programming features, shared history files, and themed prompts are a few of the extended Bourne shell components it provides.

When looking at shells, it is important to understand the history and current use of the /bin/sh file. Originally, this file was the location of the system's shell. For example, on Unix systems, you would typically find the Bourne shell installed here. On Linux systems, the /bin/sh file is now a symbolic link (covered in Chapter 4) to a shell. Typically the file points to the Bash shell (bash) as shown in Listing 1.1 on a CentOS distribution via the readlink command.

Listing 1.1: Showing to which shell /bin/sh points on a CentOS distribution

```
$ readlink /bin/sh
bash
$
```

It is always a good idea to check which shell the file is linked to. In Listing 1.2, you can see that the /bin/sh file is a symbolic link to the Dash shell (dash).

Listing 1.2: Showing to which shell /bin/sh points on an Ubuntu distribution

```
$ readlink /bin/sh
dash
$
```

To quickly determine what shell you are using at the command line, you can employ an environment variable (environment variables are covered in detail later in this chapter) along with the echo command. The echo command allows you to display data to the screen. In Listing 1.3, on a CentOS distro, the environment variable (SHELL) has its data (the current shell program) displayed by using the echo command. The $ is added prior to the variable's name in order to tap into the data stored within that variable.

Listing 1.3: Displaying the current shell on a CentOS distribution

```
$ echo $SHELL
/bin/bash
$
$ echo $BASH_VERSION
4.2.46(2)-release
$
```

Notice in Listing 1.3 that the current shell is the Bash (/bin/bash) shell. You can also show the current version of the Bash shell via the BASH_VERSION environment variable as also shown in the listing.

While you are exploring your shell environment, you should learn information about your system's Linux kernel as well. The uname utility is helpful here. A few examples are shown in Listing 1.4.

Listing 1.4: Displaying the current shell on an Ubuntu distribution

```
$ uname
Linux
$
$ uname -r
4.15.0-46-generic
$
$ uname -a
Linux Ubuntu1804 4.15.0-46-generic #49-Ubuntu SMP Wed Feb 6
09:33:07 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
$
```

When used by itself, the `uname` command displays only the kernel's name (`Linux`). If you want to know the current kernel version (called the *revision*), add the `-r` command option, as shown in Listing 1.4. To see all system information this utility provides, such as the processor type (`x86_64`) and operating system name (`GNU/Linux`), tack on the `-a` option to the `uname` command.

Using a Shell

To run a program from the shell, at the command line you simply type its command, using the proper syntax, and press Enter to execute it. The `echo` command, used earlier, is a great program to start with. Its basic syntax is as follows:

```
echo [OPTION]… [STRING]…
```

In the `echo` command's syntax structure, `[OPTION]` means there are various options (also called *switches*) you can add to modify the display. The brackets indicate that switches are optional. The `[STRING]` argument allows you to designate a string to display. It too is optional, as denoted by the brackets.

An example is helpful to understand the `echo` command's syntax. In Listing 1.5 the `echo` command is employed with no arguments or options and simply displays a blank line. In its second use, the command shows the string provided as an argument to the `echo` program.

Listing 1.5: Using the `echo` command

```
$ echo

$ echo I Love Linux
I Love Linux
$
```

Quoting Metacharacters

The echo command is handy to demonstrate another useful shell feature: shell quoting. Within the Bash shell are several characters that have special meanings and functions. These characters are called *metacharacters*. Bash shell metacharacters include the following:

```
* ? [ ] ' " \ $ ; & ( ) | ^ < >
```

For example, the dollar sign ($) often indicates that the characters following it are a variable name. When used with the echo command, the program will attempt to retrieve the variable's value and display it. An example is shown in Listing 1.6.

Listing 1.6: Using the echo command with a $ metacharacter

```
$ echo $SHELL
/bin/bash
$
$ echo It cost $1.00
It cost .00
$
```

Due to the $ metacharacter, the echo command treats both the $SHELL and $1 as variables. Since $1 is not a variable in the second echo command and has no value, in its output echo displays nothing for $1. To fix this problem, you can employ *shell quoting*. Shell quoting allows you to use metacharacters as regular characters. To shell quote a single character, use the backslash (\) as shown in Listing 1.7.

Listing 1.7: Using the echo command and shell quoting a single metacharacter

```
$ echo It cost \$1.00
It cost $1.00
$
$ echo Is Schrodinger\'s cat alive or dead\?
Is Schrodinger's cat alive or dead?
$
```

While the backslash is handy for shell quoting a single metacharacter, it can be tiresome when you have multiple ones. For several metacharacters, consider surrounding them with either single or double quotation marks, depending on the situation. A few examples of this shell quoting method are shown in Listing 1.8.

Listing 1.8: Using the echo command and shell quoting multiple metacharacters

```
$ echo Is "Schrodinger's" cat alive or "dead?"
Is Schrodinger's cat alive or dead?
$
```

```
$ echo "Is Schrodinger's cat alive or dead?"
Is Schrodinger's cat alive or dead?
$
$ echo 'Is Schrodinger's cat alive or dead?'
> ^C
$
```

Notice the first two quoting methods work in Listing 1.8. The last one does *not* work. When the metacharacter to be shell quoted is a single quotation mark, you will need to employ another shell quoting method, such as the backslash or double quotation marks.

Navigating the Directory Structure

Files on a Linux system are stored within a single directory structure, called a *virtual directory*. The virtual directory contains files from all the computer's storage devices and merges them into a single directory structure. This structure has a single base directory called the *root directory*, often simply called *root*.

When you log into the Linux system, your process's *current working directory* is your account's *home directory*. A current working directory is the directory your process is currently using within the virtual directory structure. Think of the current working directory as the room you are currently in within your home.

You can navigate through this virtual directory structure via the `cd` command. A helpful partner to `cd` is the `pwd` command. The `cd` command moves your working directory to a new location in the virtual directory structure, and the `pwd` program displays (prints) the current working directory. A few examples are shown in Listing 1.9.

Listing 1.9: Using the `cd` and `pwd` commands

```
$ pwd
/home/Christine
$
$ cd /etc
$ pwd
/etc
$
```

It is vital to know your current working directory, especially as you traverse the virtual structure. By default, many shell commands operate on the current working directory.

When using the `cd` command, you can employ either absolute or relative directory references. *Absolute directory references* always begin with a forward slash (`/`) in reference to the root directory and use the full name of the directory location. Listing 1.9 contains absolute directory references.

Relative directory references allow you to move with the cd command to a new position in the directory structure in relation to your current working directory using shorter directory arguments. Examples of relative moves are shown in Listing 1.10.

Listing 1.10: Using the cd command with relative directory references

```
$ pwd
/etc
$
$ cd cups
$ pwd
/etc/cups
$
$ cd ..
$ pwd
/etc
$
```

In Listing 1.10 the current working directory is /etc. By issuing the cd cups command, which is a relative directory reference, you change the current working directory to /etc/cups. Notice that pwd always displays the directory using an absolute directory reference. The last command (cd ..) is also a relative move. The two dots (..) represent the directory above the current directory, which is the parent directory.

You can also employ the single dot (.) directory reference, which refers to the current working directory. Although the single dot is not used with the cd command, it is commonly employed for tasks such as copying or moving files.

The cd command has several other shortcuts you can employ besides just the two dots. For example, to change your current working directory to your user account's home directory, use one of the following:

- cd
- cd ~
- cd $HOME

You can also quickly return to your most recent working directory by employing the cd - shortcut command. An example is shown in Listing 1.11.

Listing 1.11: Using the cd - shortcut command

```
$ pwd
/etc
$
```

```
$ cd /var
$ pwd
/var
$
$ cd -
/etc
$ pwd
/etc
$
```

Understanding Internal and External Commands

Within a shell, some commands that you type at the command line are part of (internal to) the shell program. These internal commands are sometimes called *built-in commands*. Other commands are external programs, because they are not part of the shell.

You can tell whether a command is an internal or external program via the `type` command. A few examples are shown in Listing 1.12.

Listing 1.12: Using `type` to determine whether a command is external or internal

```
$ type echo
echo is a shell builtin
$
$ type pwd
pwd is a shell builtin
$
$ type uname
uname is /usr/bin/uname
$
```

Notice in Listing 1.12 that both the `echo` and `pwd` commands are internal (built-in) programs. However, the `uname` command is an external program, which is indicated by the `type` command displaying the `uname` program's absolute directory reference within the virtual directory structure.

A command may be available both internally and externally to the shell. In this case, it is important to know their differences, because they may produce slightly different results or require different options.

Using Environment Variables

Environment variables track specific system information, such as the name of the user logged into the shell, the default home directory for the user, the search path the shell uses

to find executable programs, and so on. Table 1.1 shows some of the more commonly used environment variables.

TABLE 1.1 Commonly used environment variables

Name	Description
BASH_VERSION	Current Bash shell instance's version number (Chapter 1)
EDITOR	Default editor used by some shell commands (Chapter 1)
GROUPS	User account's group memberships (Chapter 7)
HISTFILE	Name of the user's shell command history file (Chapter 1)
HISTSIZE	Maximum number of commands stored in history file (Chapter 1)
HOME	Current user's home directory name (Chapter 1)
HOSTNAME	Current system's host name (Chapter 8)
LANG	Locale category for the shell (Chapter 6)
LC_*	Various locale settings that override LANG (Chapter 6)
LC_ALL	Locale category for the shell that overrides LANG (Chapter 6)
LD_LIBRARY_PATH	Colon-separated list of library directories to search prior to looking through the standard library directories (Chapter 2)
PATH	Colon-separated list of directories to search for commands (Chapter 1)
PS1	Primary shell command-line interface prompt string (Chapter 1)
PS2	Secondary shell command-line interface prompt string
PWD	User account's current working directory (Chapter 1)
SHLVL	Current shell level (Chapter 1)
TZ	User's time zone, if different from system's time zone (Chapter 6)
UID	User account's user identification number (Chapter 7)
VISUAL	Default screen-based editor used by some shell commands (Chapter 1)

You can display a complete list of active environment variables available in your shell by using the `set` command, as shown snipped in Listing 1.13.

Listing 1.13: Using `set` to display active environment variables

```
$ set
[…]
BASH=/bin/bash
[…]
HISTFILE=/home/Christine/.bash_history
[…]
HISTSIZE=1000
HOME=/home/Christine
HOSTNAME=localhost.localdomain
[…]
PS1='$ '
PS2='> '
[…]
SHELL=/bin/bash
[…]
$
```

Besides the `set` utility, you can also employ the `env` and `printenv` commands to display variables. The `env` and `printenv` utilities allow you to see locally defined variables, such as those created in a shell script (covered in Chapter 9) as well as environment variables.

When you enter a program name (command) at the shell prompt, the shell will search all the directories listed in the `PATH` environment variable for that program. If the shell cannot find the program, you will receive a `command not found` error message. Listing 1.14 shows an example.

Listing 1.14: Viewing how `PATH` affects command execution

```
$ echo $PATH
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:
/home/Christine/.local/bin:/home/Christine/bin
$
$ ls /home/Christine/Hello.sh
/home/Christine/Hello.sh
$
$ Hello.sh
bash: Hello.sh: command not found…
$
```

Notice in Listing 1.14 that program Hello.sh is in a directory that is not in the PATH environment variable's directories. Thus, when the Hello.sh program name is entered at the shell prompt, the command not found error message is displayed.

To run a program that does not reside in a PATH directory location, you must provide the command's absolute directory reference when entering the program's name at the command line, as shown in Listing 1.15.

Listing 1.15: Executing a program outside the PATH directories

```
$ /home/Christine/Hello.sh
Hello World
$
```

The which utility is helpful in these cases. It searches through the PATH directories to find the program. If it locates the program, it displays its absolute directory reference. This saves you from having to look through the PATH variable's output yourself, as Listing 1.16 shows.

Listing 1.16: Using the which utility

```
$ which Hello.sh
/usr/bin/which: no Hello.sh in (/usr/local/bin:/usr/bin:
/usr/local/sbin:/usr/sbin:/home/Christine/.local/bin:/home/Christine/bin)
$
$ which echo
/usr/bin/echo
$
```

Notice that the which utility does not find the Hello.sh program and displays all the PATH directories it searched. However, it does locate the echo command.

If a program resides in a PATH directory, you can run it by simply entering the command's name. If desired, you can also execute it by including its absolute directory reference, as shown in Listing 1.17.

Listing 1.17: Using different references to run a command

```
$ echo Hello World
Hello World
$
$ /usr/bin/echo Hello World
Hello World
$
```

You can modify environment variables. An easy one to change is the variable controlling your shell prompt (PS1). An example of changing this variable is shown in Listing 1.18.

Listing 1.18: Setting the PS1 variable

```
$ PS1="My new prompt: "
My new prompt:
```

Notice that to change the environment variable, you simply enter the variable's name, followed by an equal sign (=), and then type the new value. Because the PS1 variable controls the shell prompt, the effect of changing it shows immediately.

However, you can run into problems if you use that simple method of modifying an environment variable. For example, the setting will not survive entering into a subshell. A *subshell* (sometimes called a child shell) is created when you perform certain tasks, such as running a shell script (covered in Chapter 9) or running particular commands.

You can determine whether your process is currently in a subshell by looking at the data stored in the SHLVL environment variable. A 1 indicates you are *not* in a subshell, because subshells have higher numbers. Thus, if SHLVL contains a number higher than 1, this indicates you're in a subshell.

The bash command automatically creates a subshell, which is helpful for demonstrating the temporary nature of employing the simple environment variable modification method, shown in Listing 1.19.

Listing 1.19: Demonstrating a subshell's effect on the PS1 variable

```
My new prompt: echo $SHLVL
1
My new prompt: bash
$
$ echo $PS1
$
$ echo $SHLVL
2
$ exit
exit
My new prompt:
```

Notice that the SHLVL environment variable is set to 1, until a subshell is entered via the bash command. Also note that the PS1 environment variable controlling the prompt does not survive entering into a subshell.

To preserve an environment variable's setting, you need to employ the export command. You can either use export when typing in the original variable definition, as shown in Listing 1.20, or use it after the variable is defined, by typing **export *variable-name*** at the command-line prompt.

Listing 1.20: Using `export` to preserve an environment variable's definition

```
My new prompt: export PS1="KeepPrompt: "
KeepPrompt:
KeepPrompt: bash
KeepPrompt:
KeepPrompt: echo $SHLVL
2
KeepPrompt:
KeepPrompt: PS1="$ "
$ export PS1
$
```

Notice in Listing 1.20 that the first method used the `export` command on the same line as the `PS1` environment variable setting and that the definition survives the subshell. The second change to the `PS1` variable defines it first and then employs the `export` command, so it too will survive a subshell.

If a variable is originally set to nothing (blank), such as is typically the `EDITOR` environment variable, you can simply reverse any modifications you make to the variable by using the `unset` command. An example of this is shown in Listing 1.21.

Listing 1.21: Using the `unset` command

```
$ echo $EDITOR

$ export EDITOR=nano
$
$ echo $EDITOR
nano
$
$ unset EDITOR
$
$ echo $EDITOR

$
```

Use caution when employing the `unset` command. If you use it on environment variables, such as `PS1`, you can cause confusing things to happen. If the variable had a different definition before you modified it, it is best to change it back to its original setting instead of using `unset` on the variable.

Getting Help

When using the various utilities at the command line, sometimes you need a little extra help on a command's options or syntax. While a search engine is useful, there may be times you cannot access the Internet. Fortunately, Linux systems typically have the *man pages* installed locally. The man pages contain documentation on a command's purpose, various options, program syntax, and so on. This information is often created by the programmer who wrote the utility.

To access a man page for a particular program—for example, the `uname` command—just type in **`man uname`** at the command line. This text-based help system will take over the entire terminal display, allowing you to read through the documentation for the chosen command.

By default, the man pages use the `less` pager utility (covered later in this chapter), allowing you to go back and forth through the display using either the PageUp or PageDown key. You can also employ the arrow keys or the spacebar as desired.

If you use the man pages to read through a built-in command's documentation, you'll reach the General Commands Manual page for Bash built-ins. It can be tedious using keys to find a command in this page. Instead, type **/** and follow it with the command name. You may have to do this two or three times to reach the utility's documentation, but it is much faster than continually pressing arrow or PageDown keys.

A handy feature of the `man` utility is the ability to search for keywords in the documentation. Just employ the `-k` option as shown snipped in Listing 1.22.

Listing 1.22: Using the `man -k` command to search for keywords

```
$ man -k passwd
[…]
passwd (1)           - update user's authentication tokens
[…]
passwd (5)           - password file
[…]
smbpasswd (5)        - The Samba encrypted password file
[…]
$
```

Instead of `man -k`, you can use the `apropos` command. For example, enter **`apropos passwd`** at the command line. However, `man -k` is easier to type.

Notice in Listing 1.22 that several items are found in the man pages using the keyword search. Next to the utility name, the man page section number is displayed in parentheses. The man pages have nine sections that contain various types of documentation. Type **`man man`** at the command line to view help information on using the man pages as well as the different section names.

Although it is a nice feature that the man pages contain more than just documentation on commands, it can cause you problems if a utility has the same name as other items, such as a file in the case of `passwd`. Typically the `man` command follows a predefined search order and, in the case of duplicate names, will show you only the utility's man page. If you need to see a different page, use the section number along with the item's name. There are a few methods for doing this, using as an example the `passwd` file documented in section 5:

```
man -S 5 passwd
man -s 5 passwd
man 5 passwd
```

If you use the `man` utility and receive a message similar to `nothing appropriate`, first check the spelling of the search term. If the spelling is correct, it's possible that the man utility's database has not been updated. You'll need to use super user privileges and issue the `makewhatis` command (on older Linux distributions) or the `mandb` command.

Besides getting help from the man pages, you can get help from your *command-line history*. The shell keeps track of all the commands you have recently used and stores them in your login session's history list. To see your history list, enter **`history`** at the shell prompt, as shown snipped in Listing 1.23.

Listing 1.23: Using the `history` command to view recent commands

```
$ history
[…]
915  echo $EDITOR
916  export EDITOR=nano
917  echo $EDITOR
918  unset EDITOR
919  echo $EDITOR
920  man -k passwd
[…]
$
```

Notice that each command is preceded by a number. This allows you to recall a command from your history list via its number and have it automatically executed, as shown snipped in Listing 1.24.

Listing 1.24: Reexecuting commands in the command history

```
$ !920
man -k passwd
[…]
passwd (1)           - update user's authentication tokens
[…]
passwd (5)           - password file
[…]
$
```

Note that in order to rerun the command, you must put an exclamation mark (`!`) prior to the number. The shell will display the command you are recalling and then execute it, which is handy.

If the `history` command does not work for you, your user account may be using a different shell than the Bash shell. You can quickly check by entering **`echo $SHELL`** at the command line. If you do not see `/bin/bash` displayed, that is a problem. Modify your user account to use `/bin/bash` as your default shell (modifying users' accounts is covered in Chapter 7). You'll need to log out and back in again for the change to take effect.

To reexecute your most recent command, enter **`!!`** at the command line and press Enter. A faster alternative is to press the up arrow key and then press Enter. Another advantage of this last method is that you can edit the command as needed prior to running it.

The history list is preserved between login sessions in the file designated by the `$HISTFILE` environment variable. It is typically the `.bash_history` file in your home directory, as shown in Listing 1.25.

Listing 1.25: Viewing the history filename

```
$ echo $HISTFILE
/home/Christine/.bash_history
$
```

Keep in mind that the history file will not have commands you have used during your current login session. These commands are stored only in the history list.

If you desire to update the history file or the current history list, you'll need to issue the `history` command with the correct option. The following is a brief list of `history` options to help you make the right choice:

- `-a` appends the current history list commands to the end of the history file.
- `-n` appends the history file commands from the current Bash shell session to the current history list.
- `-r` overwrites the current history list commands with the commands stored in the history file.

If you want to remove your command-line history, it is fairly easy to do. First, clear your current history list by typing **`history -c`** at the command line. After that, wipe the history file by issuing the **`history -w`** command, which copies the now blank history list to the `.bash_history` file, overwriting its contents.

Editing Text Files

Manipulating text is performed on a regular basis when managing a Linux system. Whether you need to modify a configuration file or create a shell script, being able to use an interactive text file editor at the command line is an important skill.

Looking at Text Editors

Three popular Linux command-line text editors are

- `emacs`
- `nano`
- `vim`

The nano editor is a good text editor to start using if you have never dealt with an editor or have used only GUI editors. To start using the nano text editor, type **nano** followed by the file's name you wish to edit or create. Figure 1.1 shows a nano text editor in action, editing a file named `numbers.txt`.

FIGURE 1.1 Using the nano text editor

The shortcut list is one of the nano text editor's most useful features. This list at the window's bottom displays the most common commands and their associated shortcut keys. The caret (^) symbol in this list indicates that the Ctrl key must be used. For example, to move down a page, you press and hold the Ctrl key and then press the V key. To see additional commands, press the Ctrl+G key combination for help.

Within the nano text editor's help subsystem, you'll see some key combinations denoted by M-k. An example is M-W for repeating a search. These are metacharacter key combinations, and the M represents the Esc, Alt, or Meta key, depending on your keyboard's setup. The k simply represents a keyboard key, such as W.

The nano text editor is wonderful to use for simple text file modifications. However, if you need a more powerful text editor for creating programs or shell scripts, popular choices include the emacs and the vim editor.

Real World Scenario

Dealing with Default Editors

Some utilities such as crontab (covered in Chapter 9) use a default editor (also called a *standard editor*) such as vim. If you are new to text editing, you may prefer to use a text editor that is fairly easy to use, so being forced to use an advanced editor is problematic.

You can change your account's standard editor via the EDITOR and VISUAL environment variables. The EDITOR variable was originally for line-based editors, such as the old ed utility. The VISUAL variable is for screen-based editors (text editors that take up the whole screen, such as nano, emacs, and vim).

Change your standard editor to your desired editor by typing, for example, **export EDITOR=nano** at the command line. Do the same for the VISUAL environment variable. Even better, add these lines to an environment file (covered in Chapter 9) so that they are set up automatically for you each time you log into the Linux system.

To start using the emacs text editor, type **emacs** followed by the file's name you wish to edit or create. Figure 1.2 shows an emacs text editor screen editing a newly created file named MyFile.txt.

FIGURE 1.2 Using the emacs text editor

Adding and modifying text, as well as moving around this editor, is fairly straightforward. However, to tap into the power of the `emacs` editor, you need to learn the various shortcut keystrokes. Here are a few examples:

- Press the Ctrl+X and then the Ctrl+S key combinations to save the editor buffer's contents to the file.
- Press the Ctrl+X and then the Ctrl+C key combinations to leave the editor.
- Press the Ctrl+H key combination and then the T key to reach the `emacs` tutorial.

Note that in the `emacs` editor documentation the Ctrl key is represented by a single C letter, and to add an additional key to it, the documentation uses a hyphen (-), instead of the traditional plus sign (+).

Though `emacs` commands are a little tricky as you begin using this editor, the benefits of learning the `emacs` editor include the following:

- Editing commands used in `emacs` can also be used to quickly edit your commands entered at the shell's command line.
- The `emacs` editor has a GUI counterpart with all the same editing features.
- You can focus on the editor's features you need most and learn its advanced capabilities later.

The `emacs` text editor is typically not installed by default. Installing software is covered in Chapter 2. Its software package name is also `emacs`.

Before we take a look at using the `vim` editor, we need to talk about `vim` versus `vi`. The `vi` editor was a Unix text editor, and when it was rewritten as an open source tool, it was improved. Thus, `vim` stands for "`vi` improved."

Often you'll find the `vi` command will start the `vim` editor. In other distributions, only the `vim` command will start the `vim` editor. Sometimes both commands work. Listing 1.26 demonstrates using the `which` utility to determine what command a CentOS distribution is using.

Listing 1.26: Using `which` to determine the editor command

```
$ which vim
/usr/bin/vim
$
$ which vi
alias vi='vim'
/usr/bin/vim
$
```

Listing 1.26 shows that this CentOS distribution has aliased the `vi` command to point to the `vim` command. Thus, for this distribution both the `vi` and `vim` commands will start the `vim` editor.

Some distributions, such as Ubuntu, do not have the `vim` editor installed by default. Instead, they use an alternative, called `vim.tiny`, which will not allow you to try out all the various `vim` commands discussed here. You can check your distribution to see if `vim` is installed by obtaining the `vim` program filename. Type **`type vi`** and press Enter, and if you get an error or an alias, then enter **`type vim`**. After you receive the program's directory and filename, type the command **`readlink -f`** and follow it up with the directory and filename—for example, `readlink -f /usr/bin/vi`. If you see `/usr/bin/vi.tiny`, you need to either switch to a different distribution to practice the `vim` commands or install the `vim` package (see Chapter 2).

To start using the `vim` text editor, type **`vim`** or **`vi`**, depending on your distribution, followed by the name of the file you wish to edit or create. Figure 1.3 shows a `vim` text editor screen in action.

FIGURE 1.3 Using the `vim` text editor

In Figure 1.3 the file being edited is named `numbers.txt`. The `vim` editor works the file data in a memory buffer, and this buffer is displayed on the screen. If you open `vim` without a filename or the filename you entered doesn't yet exist, `vim` starts a new buffer area for editing.

The `vim` editor has a message area near the bottom line. If you have just opened an already created file, it will display the filename along with the number of lines and characters read into the buffer area. If you are creating a new file, you will see `[New File]` in the message area.

Understanding *vim* Modes

The `vim` editor has three standard modes as follows:

Command Mode This is the mode `vim` uses when you first enter the buffer area; it is sometimes called normal mode. Here you enter keystrokes to enact commands. For example, pressing the J key will move your cursor down one line. Command is the best mode to use for quickly moving around the buffer area.

Insert Mode Insert mode is also called edit or entry mode. This is the mode where you can perform simple editing. There are not many commands or special mode keystrokes. You enter this mode from command mode by pressing the I key. At this point, the message `--Insert--` will display in the message area. You leave this mode by pressing the Esc key.

Ex Mode This mode is sometimes also called colon commands because every command entered here is preceded with a colon (`:`). For example, to leave the `vim` editor and not save any changes you type **`:q`** and press the Enter key.

Exploring Basic Text-Editing Procedures

Since you start in command mode when entering the `vim` editor's buffer area, it's good to understand a few of the commonly used commands to move around in this mode. Table 1.2 contains several moving commands.

TABLE 1.2 Commonly used `vim` command mode moving commands

Keystroke(s)	Description
h	Move cursor left one character.
l	Move cursor right one character.
j	Move cursor down one line (the next line in the text).
k	Move cursor up one line (the previous line in the text).

Keystroke(s)	Description
w	Move cursor forward one word to front of next word.
e	Move cursor to end of current word.
b	Move cursor backward one word.
^	Move cursor to beginning of line.
$	Move cursor to end of line.
gg	Move cursor to the file's first line.
G	Move cursor to the file's last line.
*n*G	Move cursor to file line number *n*.
Ctrl+B	Scroll up almost one full screen.
Ctrl+F	Scroll down almost one full screen.
Ctrl+U	Scroll up half of a screen.
Ctrl+D	Scroll down half of a screen.
Ctrl+Y	Scroll up one line.
Ctrl+E	Scroll down one line.

If you have a large text file and need to search for something, there are keystrokes in command mode to do that as well. Type **?** to start a forward search or **/** to start a backward search. The keystroke will display at the `vim` editor's bottom and allow you to type the text to find. If the first item found is not what you need, press Enter, and then keep pressing the n key to move to the next matching text pattern.

Quickly moving around in the `vim` editor buffer is useful. However, there are also several editing commands that help to speed up your modification process. Table 1.3 lists the more commonly used command mode editing commands. Pay close attention to each letter's case, because lowercase keystrokes often perform different operations than uppercase keystrokes.

TABLE 1.3 Commonly used `vim` command mode editing commands

Keystroke(s)	Description
a	Insert text after cursor.
A	Insert text at end of text line.
dd	Delete current line.
dw	Delete current word.
i	Insert text before cursor.
I	Insert text before beginning of text line.
o	Open a new text line below cursor, and move to insert mode.
O	Open a new text line above cursor, and move to insert mode.
p	Paste copied text after cursor.
P	Paste copied (yanked) text before cursor.
yw	Yank (copy) current word.
yy	Yank (copy) current line.

In command mode, you can take the editing commands a step further by using their full syntax, which is as follows:

```
COMMAND [NUMBER-OF-TIMES] ITEM
```

For example, if you wanted to delete three words, you would press the D, 3, and W keys. If you wanted to copy (yank) the text from the cursor to the end of the text line, you would press the Y $ keys, move to the location you desired to paste the text, and press the P key.

Keep in mind that some people stay in command mode to get where they need to be within a file and then press the I key to jump into insert mode for easier text editing. This is a convenient method to employ.

The third `vim` mode, Ex mode, has additional handy commands. You must be in command mode to enter into Ex mode. You cannot jump from insert mode to Ex mode. Therefore, if you're currently in insert mode, press the Esc key to go back to command mode first.

Table 1.4 shows a few Ex commands that can help you manage your text file. Notice that all the keystrokes include the necessary colon (:) to use Ex commands.

TABLE 1.4 Commonly used vim Ex mode commands

Keystrokes	Description
:! *command*	Execute shell *command* and display results, but don't quit editor.
:r! *command*	Execute shell *command* and include the results in editor buffer area.
:r *file*	Read *file* contents and include them in editor buffer area.

Saving Changes

After you have made any needed text changes in the vim buffer area, it's time to save your work. You can use one of many methods as shown in Table 1.5. Type **ZZ** in command mode to write the buffer to disk and exit your process from the vim editor.

TABLE 1.5 Saving changes in the vim text editor

Mode	Keystrokes	Description
Ex	:x	Write buffer to file and quit editor.
Ex	:wq	Write buffer to file and quit editor.
Ex	:wq!	Write buffer to file and quit editor (overrides protection).
Ex	:w	Write buffer to file and stay in editor.
Ex	:w!	Write buffer to file and stay in editor (overrides protection).
Ex	:q	Quit editor without writing buffer to file.
Ex	:q!	Quit editor without writing buffer to file (overrides protection).
Command	ZZ	Write buffer to file and quit editor.

After reading through the various mode commands, you may see why some people despise the vim editor. There are a lot of obscure commands to know. However, some people love the vim editor because it is so powerful.

Some distributions have a `vim` tutorial installed by default. This is a handy way to learn to use the `vim` editor. To get started, just type **`vimtutor`** at the command line. If you need to leave the tutorial before it is complete, just type the Ex mode command **`:q`** to quit.

It's tempting to learn only one text editor and ignore the others. Knowing at least two text editors is useful in your day-to-day Linux work. For simple modifications, the `nano` text editor shines. For more complex editing, the `vim` and `emacs` editors are preferred. All are worth your time to master.

Processing Text Using Filters

At the Linux command line, you often need to view files or portions of them. In addition, you may need to employ tools that allow you to gather data chunks or file statistics for troubleshooting or analysis purposes. The utilities in this section can assist in all these activities.

File-Combining Commands

Putting together short text files for viewing on your screen and comparing them is useful. The file-combining commands covered here will do just that.

The basic utility for viewing entire text files is the concatenate command. Though this tool's primary purpose in life is to join together text files and display them, it is often used just to display a single small text file. To view a small text file, use the `cat` command with the basic syntax that follows:

```
cat [OPTION]… [FILE]…
```

The `cat` command is simple to use. You just enter the command followed by any text file you want to read, such as shown in Listing 1.27.

Listing 1.27: Using the `cat` command to display a file

```
$ cat numbers.txt
42
2A
52
0010 1010
*
$
```

The `cat` command spits out the entire text file to your screen. When you get your prompt back, you know that the line above the prompt is the file's last line.

There is a handy new clone of the `cat` command called `bat`. Its developer calls it "`cat` with wings," because of the `bat` utility's many additional features. You can read about its features at `github.com/sharkdp/bat`.

In Listing 1.28 is an example of concatenating two files together to display their text contents one after the other using the `cat` command.

Listing 1.28: Using the `cat` command to concatenate files

```
$ cat numbers.txt random.txt
42
2A
52
0010 1010
*
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
```

Both of the files displayed in Listing 1.28 have the number `42` as their first line. This is the only way you can tell where one file ends and the other begins, because the `cat` utility does not denote a file's beginning or end in its output.

Unfortunately, often the `cat` utility's useful formatting options go unexplored. Table 1.6 has a few of the more commonly used switches.

TABLE 1.6 The `cat` command's commonly used options

Short	Long	Description
`-A`	`--show-all`	Equivalent to using the option `-vET` combination.
`-E`	`--show-ends`	Display a $ when a newline linefeed is encountered.
`-n`	`--number`	Number all text file lines and display that number in the output.
`-s`	`--squeeze-blank`	Do not display repeated blank empty text file lines.
`-T`	`--show-tabs`	Display a `^I` when a tab character is encountered.
`-v`	`--show-nonprinting`	Display nonprinting characters when encountered using either ^ and/or M- notation.

Being able to display nonprinting characters with the `cat` command is handy. If you have a text file that is causing some sort of odd problem when processing it, you can quickly see if there are any nonprintable characters embedded. In Listing 1.29 an example is shown of this method.

Listing 1.29: Using the `cat` command to display nonprintable characters

```
$ cat bell.txt

$ cat -v bell.txt
^G
$
```

In Listing 1.29, the first `cat` command displays the file, and it appears to simply contain a blank line. However, when the `-v` option is employed, you can see that a nonprintable character exists within the file. The `^G` is in caret notation and indicates that the nonprintable Unicode character BEL is embedded in the file. This character causes a bell sound when the file is displayed.

There are interesting variants of the `cat` command—`bzcat`, `xzcat`, and `zcat`. These utilities are used to display the contents of compressed files. (File compression is covered in Chapter 4.)

If you want to display two files side-by-side and you do not care how sloppy the output is, you can use the `paste` command. Just like school paste, it will glue them together, but the result will not necessarily be pretty. An example of using the `paste` command is shown in Listing 1.30.

Listing 1.30: Using the `paste` command to join together files side-by-side

```
$ cat random.txt
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
$ cat numbers.txt
42
2A
52
0010 1010
*
$
```

```
$ paste random.txt numbers.txt
42      42
Flat Land       2A
Schrodinger's Cat       52
0010 1010       0010 1010
0000 0010       *
```

If you need a nicer display than `paste` can provide, consider using the `pr` command. If the files share the same data in a particular field, you can employ the `join` command as well.

File-Transforming Commands

Looking at a file's data in different ways is helpful not only in troubleshooting but in testing as well. We'll take a look at a few helpful file- transforming commands in this section.

Uncovering with *od*

Occasionally you may need to do a little detective work with files. These situations may include trying to review a graphics file or troubleshooting a text file that has been modified by a program. The `od` utility can help, because it allows you to display a file's contents in octal (base 8), hexadecimal (base 16), decimal (base 10), and ASCII. Its basic syntax is as follows:

```
od [OPTION]... [FILE]...
```

By default `od` displays a file's text in octal. An example is shown in Listing 1.31.

Listing 1.31: Using the `od` command to display a file's text in octal

```
$ cat fourtytwo.txt
42
fourty two
quarante deux
zweiundvierzig
forti to
$
$ od fourtytwo.txt
0000000 031064 063012 072557 072162 020171 073564 005157 072561
0000020 071141 067141 062564 062040 072545 005170 073572 064545
0000040 067165 073144 062551 075162 063551 063012 071157 064564
0000060 072040 005157
0000064
```

The first column of the od command's output is an index number for each displayed line. For example, in Listing 1.31, the line beginning with 0000040 indicates that the third line starts at octal 40 (decimal 32) bytes in the file.

You can use other options to improve the "readability" of the od command's display or to view different outputs (see the man pages for additional od utility options and their presentation). Listing 1.32 is an example of using the -cb options to display the characters in the file, along with each character's octal byte location in the text file.

Listing 1.32: Using the od -cb command to display additional information

```
$ od -cb fourtytwo.txt
0000000   4   2  \n   f   o   u   r   t   y       t   w   o  \n   q   u
        064 062 012 146 157 165 162 164 171 040 164 167 157 012 161 165
0000020   a   r   a   n   t   e       d   e   u   x  \n   z   w   e   i
        141 162 141 156 164 145 040 144 145 165 170 012 172 167 145 151
0000040   u   n   d   v   i   e   r   z   i   g  \n   f   o   r   t   i
        165 156 144 166 151 145 162 172 151 147 012 146 157 162 164 151
0000060       t   o  \n
        040 164 157 012
0000064
$
```

There is a proposal on the table to add a -u option to the od command. This option would allow the display of all Unicode characters, besides just the ASCII character subset now available. This would be a handy addition, so watch for this potential utility improvement.

Separating with *split*

One nice command to use is split. This utility allows you to divide a large file into smaller chunks, which is handy when you want to quickly create a smaller text file for testing purposes. The basic syntax for the split command is as follows:

```
split [OPTION]... [INPUT [PREFIX]]
```

You can divide up a file using size, bytes, lines, and so on. The original file (INPUT) remains unchanged, and additional new files are created, depending on the command options chosen. In Listing 1.33 an example shows using the option to split up a file by its line count.

Listing 1.33: Using the split -l command to split a file by line count

```
$ cat fourtytwo.txt
42
```

```
fourty two
quarante deux
zweiundvierzig
forti to
$
$ split -l 3 fourtytwo.txt split42
$
$ ls split42*
split42aa  split42ab
$
$ cat split42aa
42
fourty two
quarante deux
$
$ cat split42ab
zweiundvierzig
forti to
$
```

Notice that to split a file by its line count, you need to employ the `-l` (lowercase L) option and provide the number of text file lines to attempt to put into each new file. In the example, the original file has five text lines, so one new file (`split42aa`) gets the first three lines of the original file, and the second new file (`split42ab`) has the last two lines. Be aware that even though you specify the new files' name (*`PREFIX`*), the `split` utility tacks additional characters, such as `aa` and `ab`, onto the names, as shown in Listing 1.33.

The `tr` command is another handy file-transforming command. It is covered later in this chapter.

File-Formatting Commands

Often to understand the data within text files, you need to reformat the data in some way. There are a couple of simple utilities you can use to do this.

Organizing with *sort*

The `sort` utility sorts a file's data. Keep in mind that it makes no changes to the original file; only the output is sorted. The basic syntax of this command is as follows:

```
sort [OPTION]... [FILE]...
```

If you want to order a file's content using the system's standard sort order, enter the `sort` command followed by the name of the file you wish to sort. Listing 1.34 shows an example of this.

Listing 1.34: Employing the `sort` command

```
$ cat alphabet.txt
Alpha
Tango
Bravo
Echo
Foxtrot
$
$ sort alphabet.txt
Alpha
Bravo
Echo
Foxtrot
Tango
$
```

If a file contains numbers, the data may not be in the order you desire using the `sort` utility. To obtain proper numeric order, add the `-n` option to the command, as shown in Listing 1.35.

Listing 1.35: Using the `sort -n` command

```
$ sort counts.txt
105
37
42
54
8
$ sort -n counts.txt
8
37
42
54
105
$
```

In Listing 1.35, notice that the first attempt to numerically order the file, using the `sort` command with no options, yields incorrect results. However, the second attempt uses the `sort -n` command, which properly orders the file numerically.

If you'd like to save the output from the `sort` command to a file, all it takes is adding the `-o` switch. For example, `sort -o newfile.txt alphabet.txt` will sort the `alphabet.txt` file and store its sorted contents in the `newfile.txt` file.

Numbering with *nl*

Another useful file-formatting command is the `nl` utility (number line utility). This little command allows you to number lines in a text file in powerful ways. It even allows you to use regular expressions (covered later in this chapter) to designate which lines to number. The `nl` command's syntax is fairly simple:

```
nl [OPTION]... [FILE]...
```

If you do not use any options with the `nl` utility, it will number only non-blank text lines. An example is shown in Listing 1.36.

Listing 1.36: Using the `nl` command to add numbers to non-blank lines

```
$ nl ContainsBlankLines.txt
     1  Alpha
     2  Tango

     3  Bravo
     4  Echo


     5  Foxtrot
$
```

If you would like all file's lines to be numbered, including blank ones, then you'll need to employ the `-ba` switch. An example is shown in Listing 1.37.

Listing 1.37: Using the `nl -ba` command to number all text file lines

```
$ nl -ba ContainsBlankLines.txt
     1  Alpha
     2  Tango
     3
     4  Bravo
     5  Echo
     6
     7
     8  Foxtrot
$
```

The sed command also allows you to format text files. However, because this utility uses regular expressions, it is covered after the regular expression section in this chapter.

File-Viewing Commands

When you operate at the command line, viewing files is a daily activity. For a short text file, using the `cat` command is sufficient. However, when you need to look at a large file or a portion of it, other commands are available that work better than `cat`, and they are covered in this section.

Using *more* or *less*

One way to read through a large text file is by using a *pager*. A pager utility allows you to view one text page at a time and move through the text at your own pace. The two most commonly used pagers are the `more` and `less` utilities.

Though rather simple, the `more` utility is a nice little pager utility. The command's syntax is as follows:

```
more [OPTION] FILE [...]
```

With `more`, you can move forward through a text file by pressing the spacebar (one page down) or the Enter key (one line down). However, you cannot move backward through a file. The utility displays at the screen's bottom how far along you are in the file. When you wish to exit from the `more` pager, you must press the Q key.

A more flexible pager is the `less` utility. Figure 1.4 shows using the `less` utility on the `/etc/nsswitch.conf` text file.

FIGURE 1.4 Using the `less` text pager

```
#
# /etc/nsswitch.conf
#
# An example Name Service Switch config file. This file should be
# sorted with the most-used services at the beginning.
#
# The entry '[NOTFOUND=return]' means that the search for an
# entry should stop if the search in the previous entry turned
# up nothing. Note that if the search failed due to some other reason
# (like no NIS server responding) then the search continues with the
# next entry.
#
# Valid entries include:
#
#       nisplus                 Use NIS+ (NIS version 3)
#       nis                     Use NIS (NIS version 2), also called YP
#       dns                     Use DNS (Domain Name Service)
#       files                   Use the local files
#       db                      Use the local database (.db) files
#       compat                  Use NIS on compat mode
#       hesiod                  Use Hesiod for user lookups
#       [NOTFOUND=return]       Stop searching if not found so far
#

# To use db, put the "db" in front of "files" for entries you want to be
# looked up first in the databases
#
# Example:
#passwd:    db files nisplus nis
#shadow:    db files nisplus nis
#group:     db files nisplus nis

passwd:     files sss
shadow:     files sss
/etc/nsswitch.conf
```

Though similar to the more utility in its syntax as well as the fact that you can move through a file a page (or line) at a time, this pager utility also allows you to move backward. Yet the less utility has far more capabilities than just that, which leads to the famous description of this pager, "less is more."

The less pager utility allows faster file traversal because it does not read the entire file prior to displaying the file's first page. You can also employ the up and down arrow keys to traverse the file as well as the spacebar to move forward a page and the Esc+V key combination to move back a page. You can search for a particular word within the file by pressing the ? key, typing in the word you want to find, and pressing Enter to search backward. Replace the ? key with the / key and you can search forward. Like the more pager, you do need to use the Q key to exit.

By default, the Linux man page utility uses less as its pager. Learning the less utility's commands will allow you to search through various manual pages with ease.

The less utility has amazing capabilities. It would be well worth your time to peruse the less pager's man pages and play around using its various file search and traversal commands on a large text file.

Looking at files with *head*

Another handy tool for displaying portions of a text file is the head utility. The head command's syntax is shown as follows:

```
head [OPTION]... [FILE]...
```

By default, the head command displays the first 10 lines of a text file. An example is shown in Listing 1.38.

Listing 1.38: Employing the head command

```
$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
$
```

A good command option to try allows you to override the default behavior of only displaying a file's first 10 lines. The switch to use is `-n` (or `--lines=`), followed by an argument. The argument determines the number of file lines to display, as shown in Listing 1.39.

Listing 1.39: Using the `head` command to display fewer lines

```
$ head -n 2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
$ head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
```

Notice in Listing 1.38 that the `-n 2` switch and argument used with the `head` command display only the file's first two lines. However, the second command eliminates the `n` portion of the switch, and the command behaves just the same as the first command.

Viewing Files with *tail*

If you want to display a file's last lines instead of its first lines, employ the `tail` utility. Its general syntax is similar to the `head` command's syntax and is shown as follows:

```
tail [OPTION]... [FILE]...
```

By default, the `tail` command will show a file's last 10 text lines. However, you can override that behavior by using the `-n` (or `--lines=`) switch with an argument. The argument tells `tail` how many lines from the file's bottom to display. If you add a plus sign (`+`) in front of the argument, the `tail` utility will start displaying the file's text lines starting at the designated line number to the file's end. There are three examples of using `tail` in these ways in Listing 1.40.

Listing 1.40: Employing the `tail` command

```
$ tail /etc/passwd
saslauth:x:992:76:Saslauthd user:/run/saslauthd:/sbin/nologin
pulse:x:171:171:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
setroubleshoot:x:991:985::/var/lib/setroubleshoot:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
sssd:x:990:984:User for sssd:/:/sbin/nologin
gnome-initial-setup:x:989:983::/run/gnome-initial-setup/:/sbin/nologin
```

```
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
$ tail -n 2 /etc/passwd
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
$ tail -n +42 /etc/passwd
gnome-initial-setup:x:989:983::/run/gnome-initial-setup/:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/nologin
$
```

One of the most useful `tail` utility features is its ability to watch log files. Log files typically have new messages appended to the file's bottom. Watching new messages as they are added is very handy. Use the `-f` (or `--follow`) switch on the `tail` command and provide the log filename to watch as the command's argument. You will see a few recent log file entries immediately. As you keep watching, additional messages will display as they are being added to the log file.

Some log files have been replaced on various Linux distributions, and now the messages are kept in a journal file managed by `journald`. To watch messages being added to the journal file, use the `journalctl --follow` command.

To end your monitoring session using `tail`, you must use the Ctrl+C key combination. An example of watching a log file using the `tail` utility is shown snipped in Listing 1.41.

Listing 1.41: Watching a log file with the `tail` command

```
$ sudo tail -f /var/log/auth.log
[sudo] password for Christine:
Aug 27 10:15:14 Ubuntu1804 sshd[15662]: Accepted password […]
Aug 27 10:15:14 Ubuntu1804 sshd[15662]: pam_unix(sshd:sess[…]
Aug 27 10:15:14 Ubuntu1804 systemd-logind[588]: New sessio[…]
Aug 27 10:15:50 Ubuntu1804 sudo: Christine : TTY=pts/1 ; P[…]
Aug 27 10:15:50 Ubuntu1804 sudo: pam_unix(sudo:session): s[…]
Aug 27 10:16:21 Ubuntu1804 login[10703]: pam_unix(login:se[…]
Aug 27 10:16:21 Ubuntu1804 systemd-logind[588]: Removed se[…]
^C
$
```

If you are following along on your own system with the commands in this book, your Linux distribution may not have the `/var/log/auth.log` file. Try the `/var/log/secure` file instead.

File-Summarizing Commands

Summary information is handy to have when analyzing problems and understanding your files. Several utilities covered in this section will help you in summarizing activities.

Counting with *wc*

The easiest and most common utility for determining counts in a text file is the `wc` utility. The command's basic syntax is as follows:

```
wc [OPTION]... [FILE]...
```

When you issue the `wc` command with no options and pass it a filename, the utility will display the file's number of lines, words, and bytes in that order. Listing 1.42 shows an example.

Listing 1.42: Employing the `wc` command

```
$ wc random.txt
 5  9 52 random.txt
$
```

There a few useful and commonly used options for the `wc` command. These are shown in Table 1.7.

TABLE 1.7 The `wc` command's commonly used options

Short	Long	Description
`-c`	`--bytes`	Display the file's byte count.
`-L`	`--max-line-length`	Display the byte count of the file's longest line.
`-l`	`--lines`	Display the file's line count.
`-m`	`--chars`	Display the file's character count.
`-w`	`--words`	Display the file's word count.

An interesting `wc` option for troubleshooting configuration files is the `-L` switch. Generally speaking, line length for a configuration file will be under 150 bytes, though

there are exceptions. Thus, if you have just edited a configuration file and that service is no longer working, check the file's longest line length. A longer than usual line length indicates you might have accidently merged two configuration file lines. An example is shown in Listing 1.43.

Listing 1.43: Using the `wc` command to check line length

```
$ wc -L /etc/nsswitch.conf
72 /etc/nsswitch.conf
$
```

In Listing 1.43, the file's line length shows a normal maximum line length of `72` bytes. This `wc` command switch can also be useful if you have other utilities that cannot process text files exceeding certain line lengths.

Pulling Out Portions with *cut*

To sift through the data in a large text file, it helps to quickly extract small data sections. The `cut` utility is a handy tool for doing this. It will allow you to view particular fields within a file's records. The command's basic syntax is as follows:

```
cut OPTION... [FILE]...
```

Before we delve into using this command, there are few basics to understand concerning the `cut` command. They are as follows:

Text File Records A text file record is a single-file line that ends in a newline linefeed, which is the ASCII character LF. You can see if your text file uses this end-of-line character via the `cat -E` command. It will display every newline linefeed as a `$`. If your text file records end in the ASCII character NUL, you can also use `cut` on them, but you must use the `-z` option.

Text File Record Delimiter For some of the `cut` command options to be properly used, fields must exist within each text file record. These fields are not database-style fields but instead data that is separated by some *delimiter*. A delimiter is one or more characters that create a boundary between different data items within a record. A single space can be a delimiter. The password file, `/etc/passwd`, uses colons (`:`) to separate data items within a record.

Text File Changes Contrary to its name, the `cut` command does not change any data within the text file. It simply copies the data you wish to view and displays it to you. Rest assured that no modifications are made to the file.

The `cut` utility has a few options you will use on a regular basis. These options are listed in Table 1.8.

TABLE 1.8 The `cut` command's commonly used options

Short	Long	Description
`-c` *nlist*	`--characters` *nlist*	Display only the record characters in the *nlist* (e.g., 1–5).
`-b` *blist*	`--bytes` *blist*	Display only the record bytes in the *blist* (e.g., 1–2).
`-d` *d*	`--delimiter` *d*	Designate the record's field delimiter as *d*. This overrides the Tab default delimiter. Put *d* within quotation marks to avoid unexpected results.
`-f` *flist*	`--fields` *flist*	Display only the record's fields denoted by *flist* (e.g., 1,3).
`-s`	`--only-delimited`	Display only records that contain the designated delimiter.
`-z`	`--zero-terminated`	Designate the record end-of-line character as the ASCII character NUL.

A `cut` command in action is shown in Listing 1.44.

Listing 1.44: Employing the `cut` command

```
$ head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
$
$ cut -d ":" -f 1,7 /etc/passwd
root:/bin/bash
bin:/sbin/nologin
[…]
$
```

In Listing 1.44, the `head` command displays the password file's first two lines. This text file employs colons (`:`) to delimit the fields within each record. The first use of the `cut` command designates the colon delimiter using the `-d` option. Notice the colon is encased in quotation marks to avoid unexpected results. The `-f` option specifies that only fields `1` (username) and `7` (shell) should be displayed.

Occasionally it is worthwhile to save a `cut` command's output. You can do this by redirecting standard output, which is covered later in this chapter.

Discovering Repeated Lines with *uniq*

A quick way to find repeated lines in a text file is with the `uniq` utility. Just type **`uniq`** and follow it with the filename whose contents you want to check.

The `uniq` utility will find repeated text lines only if they come right after one another. Used without any options, the command will display only unique (non-repeated) lines. An example of using this command is shown in Listing 1.45.

Listing 1.45: Using the `uniq` command

```
$ cat NonUniqueLines.txt
A
C
C
A
$
$ uniq NonUniqueLines.txt
A
C
A
$
```

Notice that in the `cat` command's output there are actually two sets of repeated lines in this file. One set is the `C` lines, and the other set is the `A` lines. Because the `uniq` utility recognizes only repeated lines that are one after the other in a text file, only one of the `C` text lines is removed from the display. The two `A` lines are still both shown.

Digesting an MD5 Algorithm

The `md5sum` utility is based on the MD5 message-digest algorithm. It was originally created to be used in cryptography. It is no longer used in such capacities due to various known vulnerabilities. However, it is still excellent for checking a file's integrity. A simple example is shown in Listing 1.46.

Listing 1.46: Using `md5sum` to check the original file

```
$ md5sum fourtytwo.txt
0ddaa12f06a2b7dcd469ad779b7c2a33  fourtytwo.txt
$
```

The `md5sum` produces a 128-bit hash value. If you copy the file to another system on your network, run the `md5sum` on the copied file. If you find that the hash values of the original and copied file match, this indicates no file corruption occurred during its transfer.

A malicious attacker can create two files that have the same MD5 hash value. However, at this point in time, a file that is not under the attacker's control cannot have its MD5 hash value modified. Therefore, it is imperative that you have checks in place to ensure that your file was not created by a third-party malicious user. An even better solution is to use a stronger hash algorithm.

Securing Hash Algorithms

The Secure Hash Algorithms (SHA) is a family of various hash functions. Though typically used for cryptography purposes, they can also be used to verify a file's integrity after it is copied or moved to another location.

Several utilities implement these various algorithms on Linux. The quickest way to find them is via the method shown in Listing 1.47. Keep in mind your particular distribution may store them in the `/bin` directory instead.

Listing 1.47: Looking at the SHA utility names

```
$ ls -1 /usr/bin/sha???sum
/usr/bin/sha224sum
/usr/bin/sha256sum
/usr/bin/sha384sum
/usr/bin/sha512sum
$
```

Each utility includes the SHA message digest it employs within its name. Therefore, `sha256sum` uses the SHA-256 algorithm. These utilities are used in a similar manner to the `md5sum` command. A few examples are shown in Listing 1.48.

Listing 1.48: Using `sha256sum` and `sha512sum` to check a file

```
$ sha256sum fourtytwo.txt
0b2b6e2d8eab41e73baf0961ec707ef98978bcd8c7
74ba8d32d3784aed4d286b  fourtytwo.txt
$
$ sha512sum fourtytwo.txt
ac72599025322643e0e56cff41bb6e22ca4fbb76b1d
7fac1b15a16085edad65ef55bbc733b8b68367723ced
3b080dbaedb7669197a51b3b6a31db814802e2f31  fourtytwo.txt
$
```

Notice in Listing 1.48 the different hash value lengths produced by the different commands. The `sha512sum` utility uses the SHA-512 algorithm, which is the best to use for security purposes and is typically employed to hash salted passwords in the `/etc/shadow` file on Linux.

You can use these SHA utilities, just like the `md5sum` program was used in Listing 1.46, to ensure a file's integrity when it is transferred. That way, file corruption is avoided as well as any malicious modifications to the file.

Using Regular Expressions

Many commands use *regular expressions*. A regular expression is a pattern template you define for a utility such as `grep`, which then uses the pattern to filter text. Employing regular expressions along with text-filtering commands expands your mastery of the Linux command line.

Using *grep*

A wonderful tool for sifting text is the `grep` command. The `grep` command is powerful in its use of regular expressions, which will help with filtering text files. But before we cover those, peruse Table 1.9 for commonly used `grep` utility options.

TABLE 1.9 The `grep` command's commonly used options

Short	Long	Description
`-c`	`--count`	Display a count of text file records that contain a *PATTERN* match.
`-d action`	`--directories=action`	When a file is a directory, if *action* is set to read, read the directory as if it were a regular text file; if *action* is set to skip, ignore the directory; and if *action* is set to recurse, act as if the - R, -r, or --recursive option was used.
`-E`	`--extended-regexp`	Designate the *PATTERN* as an extended regular expression.
`-i`	`--ignore-case`	Ignore the case in the *PATTERN* as well as in any text file records.
`-R, -r`	`--recursive`	Search a directory's contents, and for any subdirectory within the original directory tree, consecutively search its contents as well (recursively).
`-v`	`--invert-match`	Display only text files records that do *not* contain a *PATTERN* match.

The basic syntax for the grep utility is as follows:

```
grep [OPTION] PATTERN [FILE...]
```

A simple example is shown in Listing 1.49. No options are used, and the grep utility is used to search for the word root (*PATTERN*) within /etc/passwd (*FILE*).

Listing 1.49: Using a simple grep command to search a file

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$
```

Notice that the grep command returns each file record (line) that contains an instance of the *PATTERN*, which in this case was the word root.

You can also use a series of patterns stored in a file with a variation of the grep utility. An example of doing this is shown in Listing 1.50.

Listing 1.50: Using the grep command to search for patterns stored in a text file

```
$ cat accounts.txt
sshd
Christine
nfsnobody
$
$ fgrep -f accounts.txt /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
Christine:x:1001:1001::/home/Christine:/bin/bash
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
$
$ grep -F -f accounts.txt /etc/passwd
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
Christine:x:1001:1001::/home/Christine:/bin/bash
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
$
```

The patterns are stored in the accounts.txt file, which is first displayed using the cat command. Next, the fgrep command is employed, along with the -f option to indicate the file that holds the patterns. The /etc/passwd file is searched for all the patterns stored within the accounts.txt file, and the results are displayed.

Also notice in Listing 1.49 that the third command is the grep -F command. The grep -F command is equivalent to using the fgrep command, which is why the two commands produce identical results.

Understanding Basic Regular Expressions

Basic regular expressions (BREs) include characters, such as a dot followed by an asterisk (`.*`) to represent multiple characters and a single dot (`.`) to represent one character. They also may use brackets to represent multiple characters, such as `[a,e,i,o,u]` (you do *not* have to include the commas) or a range of characters, such as `[A-z]`. When brackets are employed, it is called a *bracket expression*.

To find text file records that begin with particular characters, you can precede them with a caret (`^`) symbol. For finding text file records where particular characters are at the record's end, append them with a dollar sign (`$`) symbol. Both the caret and the dollar sign symbols are called *anchor characters* for BREs, because they fasten the pattern to the beginning or the end of a text line.

You will see in documentation and technical descriptions different names for regular expressions. The name may be shortened to regex or regexp.

Using a BRE pattern is fairly straightforward with the `grep` utility. Listing 1.51 shows some examples.

Listing 1.51: Using the `grep` command with a BRE pattern

```
$ grep daemon.*nologin /etc/passwd
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
daemon:/dev/null:/sbin/nologin
[…]
$
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$
$ grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$
```

In the first snipped `grep` example within Listing 1.51, the `grep` command employs a pattern using the BRE `.*` characters. In this case, the `grep` utility will search the password file for any instances of the word `daemon` within a record and display that record if it *also* contains the word `nologin` after the word `daemon`.

The next two `grep` examples in Listing 1.51 are searching for instances of the word `root` within the password file. Notice that the one command displays two lines from the file. The next command employs the BRE `^` character and places it before the word `root`. This regular expression pattern causes `grep` to display only lines in the password file that *begin* with `root`.

If you would like to get a better handle on regular expressions, there are several good resources. Our favorite is Chapter 20 in the book *Linux Command Line and Shell Scripting Bible* by Blum and Bresnahan (Wiley, 2015).

You can also look at the man pages, section 7, on regular expressions (called *regex(7)* in the certification objectives). View this information by typing **`man 7 regex`** or **`man -S 7 regex`** at the command line.

The `-v` option is useful when auditing your configuration files with the `grep` utility. It produces a list of text file records that *do not* contain the pattern. Listing 1.52 shows an example of finding all the records in the password file that *do not* end in `nologin`. Notice that the BRE pattern puts the `$` at the end of the word. If you were to place the `$` before the word, it would be treated as a variable name instead of a BRE pattern.

Listing 1.52: Using the `grep` command to audit the password file

```
$ grep -v nologin$ /etc/passwd
root:x:0:0:root:/root:/bin/bash
sync:x:5:0:sync:/sbin:/bin/sync
[…]
Christine:x:1001:1001::/home/Christine:/bin/bash
$
```

If you need to filter out all the blank lines in a file (display only lines with text), use `grep` with the `-v` option to invert the matching pattern. Then employ the `^` and `$` anchor characters like **`grep -v ^$ filename`** at the command line.

A special group of bracket expressions are *character classes*. These bracket expressions have predefined names and could be considered bracket expression shortcuts. Their interpretation is based on the `LC_CTYPE` locale environment variable (locales are covered in Chapter 6). Table 1.10 shows the more commonly used character classes.

TABLE 1.10 Commonly used character classes

Class	Description
`[:alnum:]`	Matches any alphanumeric characters (any case), and is equal to using the `[0-9A-Za-z]` bracket expression
`[:alpha:]`	Matches any alphabetic characters (any case), and is equal to using the `[A-Za-z]` bracket expression

Class	Description
[:blank:]	Matches any blank characters, such as tab and space
[:digit:]	Matches any numeric characters, and is equal to using the [0-9] bracket expression
[:lower:]	Matches any lowercase alphabetic characters, and is equal to using the [a-z] bracket expression
[:punct:]	Matches punctuation characters, such as !, #, $, and @
[:space:]	Matches space characters, such as tab, form feed, and space
[:upper:]	Matches any uppercase alphabetic characters, and is equal to using the [A-Z] bracket expression

For using character classes with the grep command, enclose the bracketed character class in another set of brackets. An example of using grep with the digit character class is shown in Listing 1.53.

Listing 1.53: Using the grep command and a character class

```
$ cat random.txt
42
Flat Land
Schrodinger's Cat
0010 1010
0000 0010
$
$ grep [[:digit:]] random.txt
42
0010 1010
0000 0010
$
```

Notice the extra brackets needed to properly use a character class. Thus, to use [:digit:], you must type **[[:digit:]]** when employing this character class with the grep command.

If you need to search for a character in a file that has special meaning in an expression or at the command line, such as the $ anchor character, precede it with a backslash (\). This lets the grep utility know you are searching for that character and not using it in an expression.

Understanding Extended Regular Expressions

Extended regular expressions (EREs) allow more complex patterns. For example, a vertical bar symbol (|) allows you to specify two possible words or character sets to match. You can also employ parentheses to designate additional subexpressions.

Using ERE patterns can be rather tricky. A few examples employing `grep` with EREs are helpful, such as the ones shown in Listing 1.54.

Listing 1.54: Using the `grep` command with an ERE pattern

```
$ grep -E "^root|^dbus" /etc/passwd
root:x:0:0:root:/root:/bin/bash
dbus:x:81:81:System message bus:/:/sbin/nologin
$
$ egrep "(daemon|s).*nologin" /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$
```

In the first example, the `grep` command uses the `-E` option to indicate the pattern is an extended regular expression. If you did not employ the `-E` option, unpredictable results would occur. Quotation marks around the ERE pattern protect it from misinterpretation. The command searches for any password file records that start with either the word `root` or the word `dbus`. Thus, a caret (`^`) is placed prior to each word, and a vertical bar (`|`) separates the words to indicate that the record can start with either word.

In the second example in Listing 1.54, notice that the `egrep` command is employed. The `egrep` command is equivalent to using the `grep -E` command. The ERE pattern here also uses quotation marks to avoid misinterpretation and employs parentheses to issue a subexpression. The subexpression consists of a choice, indicated by the vertical bar (`|`), between the word `daemon` and the letter `s`. Also in the ERE pattern, the `.*` symbols are used to indicate there can be anything in between the subexpression choice and the word `nologin` in the text file record.

Take a deep breath. That was a lot to take in. However, as hard as BRE and ERE patterns are, they are worth using with the `grep` command to filter out data from your text files.

Using Streams, Redirection, and Pipes

One of the neat things about commands at the command line is that you can employ complex frameworks. These structures allow you to build commands from other commands, use a program's output as input to another program, put together utilities to perform custom operations, and so on.

Redirecting Input and Output

When processing and filtering text files, you may want to save the data produced. In addition, you may need to combine multiple refinement steps to obtain the information you need.

Handling Standard Output

It is important to know that Linux treats every object as a file. This includes the output process, such as displaying a text file on the screen. Each file object is identified using a *file descriptor*, an integer that classifies a process's open files. The file descriptor that identifies output from a command or script file is 1. It is also identified by the abbreviation STDOUT, which describes standard output.

By default, STDOUT directs output to your current terminal. Your process's current terminal is represented by the `/dev/tty` file.

A simple command to use when discussing standard output is the `echo` command. Issue the `echo` command along with a text string, and the text string will display to your process's STDOUT, which is typically the terminal screen. An example is shown in Listing 1.55.

Listing 1.55: Employing the `echo` command to display text to STDOUT

```
$ echo "Hello World"
Hello World
$
```

The neat thing about STDOUT is that you can redirect it via *redirection operators* on the command line. A redirection operator allows you to change the default behavior of where input and output are sent. For STDOUT, you redirect the output using the > redirection operator as shown in Listing 1.56.

Listing 1.56: Employing a STDOUT redirection operator

```
$ grep nologin$ /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$ grep nologin$ /etc/passwd > NologinAccts.txt
$
$ less NologinAccts.txt
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
[…]
$
```

In Listing 1.56, the password file is being audited for all accounts that use the `/sbin/nologin` shell via the `grep` command. The `grep` command's output is lengthy and was snipped in the listing. It would be so much easier to redirect STDOUT to a file.

This was done in Listing 1.56 by issuing the same grep command but tacking on a redirection operator, >, and a filename to the command's end. The effect was to send the command's output to the file NologinAccts.txt instead of the screen. Now the data file can be viewed using the less utility.

If you use the > redirection operator and send the output to a file that already exists, that file's current data will be deleted. Use caution when employing this operator.

To append data to a preexisting file, you need to use a slightly different redirection operator. The >> operator will append data to a preexisting file. If the file does not exist, it is created, and the outputted data is added to it. Listing 1.57 shows an example of using this redirection operator.

Listing 1.57: Using a STDOUT redirection operator to append text

```
$ echo "Nov 16, 2019" > AccountAudit.txt
$
$ wc -l /etc/passwd >> AccountAudit.txt
$
$ cat AccountAudit.txt
Nov 16, 2019
44 /etc/passwd
$
```

The first command in Listing 1.57 puts a date stamp into the AccountAudit.txt file. Because that date stamp needs to be preserved, the next command appends STDOUT to the file using the >> redirection operator. The file can continue to be appended to using the >> operator for future commands.

Redirecting Standard Error

Another handy item to redirect is standard error. The file descriptor that identifies a command or script file error is 2. It is also identified by the abbreviation STDERR, which describes standard error. STDERR, like STDOUT, is by default sent to your terminal (/dev/tty).

The basic redirection operator to send STDERR to a file is the 2> operator. If you need to append the file, use the 2>> operator. Listing 1.58 shows a snipped example of redirecting standard error.

Listing 1.58: Employing a STDERR redirection operator

```
$ grep -d skip hosts: /etc/*
grep: /etc/anacrontab: Permission denied
grep: /etc/audisp: Permission denied
```

```
[…]
$
$ grep -d skip hosts: /etc/* 2> err.txt
/etc/nsswitch.conf:#hosts:     db files nisplus nis dns
/etc/nsswitch.conf:hosts:      files dns myhostname
[…]
$
$ cat err.txt
grep: /etc/anacrontab: Permission denied
grep: /etc/audisp: Permission denied
[…]
$
```

The first command in Listing 1.58 was issued to find any files with the `/etc/` directory that contain the `hosts:` directive. Unfortunately, since the user does not have super user privileges, several permission denied error messages are generated. This clutters up the output and makes it difficult to see what files contain this directive.

To declutter the output, the second command in Listing 1.58 redirects STDERR to the `err.txt` file using the `2>` redirection operator. This makes it much easier to see what files contain the `hosts:` directive. If needed, the error messages can be reviewed because they reside now in the `err.txt` file.

Sometimes you want to send standard error and standard output to the same file. In these cases, use the `&>` redirection operator to accomplish your goal.

If you don't care to keep a copy of the error messages, you can always throw them away. This is done by redirecting STDERR to the `/dev/null` file as shown snipped in Listing 1.59.

Listing 1.59: Using a STDERR redirection operator to remove error messages

```
$ grep -d skip hosts: /etc/* 2> /dev/null
/etc/nsswitch.conf:#hosts:     db files nisplus nis dns
/etc/nsswitch.conf:hosts:      files dns myhostname
[…]
$
```

The `/dev/null` file is sometimes called the black hole. This name comes from the fact that anything you put into it, you cannot retrieve.

Regulating Standard Input

Standard input, by default, comes into your Linux system via the keyboard and/or other input devices. The file descriptor that identifies an input into a command or script file is 0. It is also identified by the abbreviation STDIN, which describes standard input.

As with STDOUT and STDERR, you can redirect STDIN. The basic redirection operator is the < symbol. The `tr` command is one of the few utilities that require you to redirect standard input. An example is shown in Listing 1.60.

Listing 1.60: Employing an STDIN redirection operator

```
$ cat Grades.txt
89 76 100 92 68 84 73
$
$ tr " " "," < Grades.txt
89,76,100,92,68,84,73
$
```

In Listing 1.60, the file `Grades.txt` contains various integers separated by a space. The second command utilizes the `tr` utility to change each space into a comma (,). Because the `tr` command requires the STDIN redirection symbol, it is also employed in the second command followed by the filename. Keep in mind that this command did not change the `Grades.txt` file. It only displayed to STDOUT what the file would look like with these changes.

It's nice to have a concise summary of the redirection operators. Therefore, we have provided one in Table 1.11.

TABLE 1.11 Commonly used redirection operators

Operator	Description
>	Redirect STDOUT to specified file. If file exists, overwrite it. If it does not exist, create it.
>>	Redirect STDOUT to specified file. If file exists, append to it. If it does not exist, create it.
2>	Redirect STDERR to specified file. If file exists, overwrite it. If it does not exist, create it.
2>>	Redirect STDERR to specified file. If file exists, append to it. If it does not exist, create it.
&>	Redirect STDOUT and STDERR to specified file. If file exists, overwrite it. If it does not exist, create it.
&>>	Redirect STDOUT and STDERR to specified file. If file exists, append to it. If it does not exist, create it.
<	Redirect STDIN from specified file into command.
<>	Redirect STDIN from specified file into command and redirect STDOUT to specified file.

Piping Data between Programs

If you really want to enact powerful and quick results at the Linux command line, you need to explore pipes. The pipe is a simple redirection operator represented by the ASCII character 124 (|), which is called the vertical bar, vertical slash, or vertical line.

Be aware that some keyboards and text display the vertical bar not as a single vertical line. Instead, it looks like a vertical double dash.

With the pipe, you can redirect STDOUT, STDIN, and STDERR between multiple commands all on one command line. Now that is powerful redirection.

The basic syntax for redirection with the pipe symbol is as follows:

```
COMMAND1 | COMMAND2 [| COMMANDN]…
```

The syntax for pipe redirection shows that the first command, `COMMAND1`, is executed. Its STDOUT is redirected as STDIN into the second command, `COMMAND2`. Also, you can pipe more commands together than just two. Keep in mind that any command in the pipeline has its STDOUT redirected as STDIN to the next command in the pipeline. Listing 1.61 shows a simple use of pipe redirection.

Listing 1.61: Employing pipe redirection

```
$ grep /bin/bash$ /etc/passwd | wc -l
3
$
```

In Listing 1.61, the first command in the pipe searches the password file for any records that end in `/bin/bash`. This is essentially finding all user accounts that use the Bash shell as their default account shell. The output from the first command in the pipe is passed as input into the second command in the pipe. The `wc -l` command will count how many lines have been produced by the `grep` command. The results show that there are only three accounts on this Linux system that have the Bash shell set as their default shell.

You can get very creative using pipe redirection. Listing 1.62 shows a command employing four different utilities in a pipeline to audit accounts using the `/sbin/nologin` default shell.

Listing 1.62: Employing pipe redirection for several commands

```
$ grep /sbin/nologin$ /etc/passwd | cut -d ":" -f 1 | sort | less
abrt
adm
avahi
bin
chrony
[…]
:
```

In Listing 1.62, the output from the grep command is fed as input into the cut command. The cut utility removes only the first field from each password record, which is the account username. The output of the cut command is used as input into the sort command, which alphabetically sorts the usernames. Finally, the sort utility's output is piped as input into the less command for leisurely perusing through the account usernames.

In cases where you want to keep a copy of the command pipeline's output as well as view it, the tee command will help. Similar to a tee pipe fitting in plumbing, where the water flow is sent in multiple directions, the tee command allows you to both save the output to a file and display it to STDOUT. Listing 1.63 contains an example of this handy command.

Listing 1.63: Employing the tee command

```
$ grep /bin/bash$ /etc/passwd | tee BashUsers.txt
root:x:0:0:root:/root:/bin/bash
user1:x:1000:1000:Student User One:/home/user1:/bin/bash
Christine:x:1001:1001::/home/Christine:/bin/bash
$
$ cat BashUsers.txt
root:x:0:0:root:/root:/bin/bash
user1:x:1000:1000:Student User One:/home/user1:/bin/bash
Christine:x:1001:1001::/home/Christine:/bin/bash
$
```

The first command in Listing 1.63 searches the password file for any user account records that end in /bin/bash. That output is piped into the tee command, which displays the output as well as saves it to the BashUsers.txt file. The tee command is handy when you are installing software from the command line and want to see what is happening as well as keep a log file of the transaction for later review.

Using *sed*

Another interesting command-line program is a *stream editor*. There are times where you will want to edit text without having to pull out a full-fledged text editor. A stream editor modifies text that is passed to it via a file or output from a pipeline. This editor uses special commands to make text changes as the text "streams" through the editor utility.

The command to invoke the stream editor is sed. The sed utility edits a stream of text data based on a set of commands you supply ahead of time. It is a very quick editor because it makes only one pass through the text to apply the modifications.

The sed editor changes data based on commands either entered into the command line or stored in a text file. The process the editor goes through is as follows:

1. Reads one text line at a time from the input stream
2. Matches that text with the supplied editor commands
3. Modifies the text as specified in the commands
4. Displays the modified text

After the sed editor matches all the specified commands against a text line, it reads the next text line and repeats the editorial process. Once sed reaches the end of the text lines, it stops.

Before looking at some sed examples, it is important to understand the command's basic syntax. It is as follows:

```
sed  [OPTIONS] [SCRIPT]… [FILENAME]
```

By default, sed will use the text from STDIN to modify it according to the specified commands. An example is shown in Listing 1.64.

Listing 1.64: Using sed to modify STDIN text

```
$ echo "I like cake." | sed 's/cake/donuts/'
I like donuts.
$
```

Notice that the text output from the echo command is piped as input into the stream editor. The sed utility's s command (substitute) specifies that if the first text string, cake, is found, it is changed to donuts in the output. Note that the entire command after sed is considered to be the SCRIPT, and it is encased in single quotation marks. Also notice that the text words are delimited from the s command, the quotation marks, and each other via the forward slashes (/).

Keep in mind that just using the s command will not change all instances of a word within a text stream. Listing 1.65 shows an example of this.

Listing 1.65: Using sed to globally modify STDIN text

```
$ echo "I love cake and more cake." | sed 's/cake/donuts/'
I love donuts and more cake.
$
$ echo "I love cake and more cake." | sed 's/cake/donuts/g'
I love donuts and more donuts.
$
```

In the first command in Listing 1.65, only the first occurrence of the word cake was modified. However, in the second command a g, which stands for global, was added to the sed script's end. This caused all occurrences of cake to change to donuts.

You can also modify text stored in a file. Listing 1.66 shows an example of this.

Listing 1.66: Using sed to modify file text

```
$ cat cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
```

```
Samantha does not like cake.
$
$ sed 's/cake/donuts/' cake.txt
Christine likes chocolate donuts.
Rich likes lemon donuts.
Tim only likes yellow donuts.
Samantha does not like donuts.
$
$ cat cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
Samantha does not like cake.
$
```

In Listing 1.66, the file contains text lines that contain the word cake. When the cake.txt file is added as an argument to the sed command, its data is modified according to the script. Notice that the data in the file is not modified. The stream editor only displays the modified text to STDOUT. You could save the modified text to another file name via a STDOUT redirection operator, if desired.

It may be tempting to think that the sed utility is operating on the text file as a whole, but it is not. The stream editor applies its commands to each text file line individually. Thus, in our previous example, if the word cake was found multiple times within a single text file line, you'd need to use the g global command to change all instances.

So far we've shown you only sed substitution commands, but you can also delete lines using the stream editor. To do so, you use the syntax of *'PATTERN*/d' for the sed command's *SCRIPT*. An example is shown in Listing 1.67. Notice the cake.txt file line that contains the word Christine is not displayed to STDOUT. It was "deleted" in the output, but it still exists within the text file.

Listing 1.67: Using sed to delete file text

```
$ sed '/Christine/d' cake.txt
Rich likes lemon cake.
Tim only likes yellow cake.
Samantha does not like cake.
$
```

You can also change an entire line of text. To accomplish this, you use the syntax of '*ADDRESS*c*NEWTEXT*' for the sed command's *SCRIPT*. The *ADDRESS* refers to the file's line

number, and the *NEWTEXT* is the different text line you want displayed. An example of this method is shown in Listing 1.68.

Listing 1.68: Using sed to change an entire file line

```
$ sed '4cI am a new line' cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
I am a new line
$
```

The stream editor has some rather useful command options. The more commonly used ones are displayed in Table 1.12.

TABLE 1.12 The sed command's commonly used options

Short	Long	Description
`-e` *script*	`--expression=script`	Add commands in *script* to text processing. The *script* is written as part of the sed command.
`-f` *script*	`--file=script`	Add commands in *script* to text processing. The *script* is a file.
`-r`	`--regexp-extended`	Use extended regular expressions in script.

A handy option to use is the `-e` option. This allows you to employ multiple scripts in the sed command. An example is shown in Listing 1.69.

Listing 1.69: Using sed `-e` to use multiple scripts

```
$ sed -e 's/cake/donuts/ ; s/like/love/' cake.txt
Christine loves chocolate donuts.
Rich loves lemon donuts.
Tim only loves yellow donuts.
Samantha does not love donuts.
$
```

Pay close attention to the syntax change in Listing 1.69. Not only is the `-e` option employed, but the script is slightly different too. Now the script contains a semicolon (;) between the two script commands. This allows both commands to be processed on the text stream.

Generating Command Lines

Creating command-line commands is a useful skill. There are several different methods you can use. One such method employs the `xargs` utility. The best thing about this tool is that you sound like a pirate when you pronounce it, but it has other practical values as well.

By piping STDOUT from other commands into the `xargs` utility, you can build command-line commands on the fly. Listing 1.70 shows an example of doing this.

Listing 1.70: Employing the `xargs` command

```
$ touch EmptyFile1.txt EmptyFile2.txt EmptyFile3.txt
$
$ ls  EmptyFile?.txt
EmptyFile1.txt  EmptyFile2.txt  EmptyFile3.txt
$
$ ls -1 EmptyFile?.txt | xargs -p /usr/bin/rm
/usr/bin/rm EmptyFile1.txt EmptyFile2.txt EmptyFile3.txt ?...n
$
```

In Listing 1.70, three blank files are created using the `touch` command. The third command uses a pipeline. The first command in the pipeline lists any files that have the name `EmptyFile`*n*`.txt`. The output from the `ls` command is piped as STDIN into the `xargs` utility. The `xargs` command uses the `-p` option. This option causes the `xargs` utility to stop and ask permission before enacting the constructed command-line command. Notice that the absolute directory reference for the `rm` command is used (the `rm` command is covered in more detail in Chapter 4). This is sometimes needed when employing `xargs`, depending on your distribution.

The created command, in Listing 1.70, attempts to remove all three empty files with one `rm` command. We typed **n** and pressed the Enter key to preserve the three files instead of deleting them, because they are needed for the next example.

Another method to created command-line commands on the fly uses shell expansion. The technique here puts a command to execute within parentheses and precedes it with a dollar sign. An example of this method is shown in Listing 1.71.

Listing 1.71: Using the $() method to create commands

```
$ rm -i $(ls EmptyFile?.txt)
rm: remove regular empty file ‘EmptyFile1.txt’? y
rm: remove regular empty file ‘EmptyFile2.txt’? y
rm: remove regular empty file ‘EmptyFile3.txt’? y
$
```

In Listing 1.71, the `ls` command is again used to list any files that have the name `EmptyFile`*n*`.txt`. Because the command is encased by the `$()` symbols, it does not display to STDOUT. Instead, the filenames are passed to the `rm -i` command, which inquires as whether or not to delete each found file. This method allows you to get very creative when building commands on the fly.

Summary

Understanding fundamental shell concepts and being able to effectively and swiftly use the right commands at the shell command line is important for your daily job. It allows you to gather information, peruse text files, filter data, and so on.

This chapter's purpose was to improve your Linux command-line tool belt. Not only will this help you in your day-to-day work life, but it will also help you successfully pass the LPI certification exam.

Exam Essentials

Express the different basic shell concepts. The shell program provides the command-line prompt, which can be reached through a tty terminal or by employing a GUI terminal emulator. There are multiple shell programs, but the most popular is the Bash shell, which is typically located in the `/bin/bash` file. The `/bin/sh` file is often linked to the Bash shell program, but it may be linked to other shells, such as the Dash shell (`/bin/dash`). The shell in use can be checked via displaying the `SHELL` environment variable's contents with the `echo` utility. The current Linux kernel can be shown with the `uname -a` command.

Summarize the various utilities that can be employed to read text files. To read entire small text files, you can use the `cat` and `bat` utilities. If you need to read only the first or last lines of a text file, employ either the `head` or `tail` command. For a single text line out of a file, the `grep` utility is useful. For reviewing a file a page at a time, you can use either the `less` or the `more` pager utility.

Describe the various methods used for editing text. Editing text files is part of a system administrator's life. You can use full-screen editors such as the rather complicated `vim` text editor or the simple and easy-to-use `nano` editor. For fast and powerful text stream editing, employ the use of `sed` and its scripts.

Summarize the various utilities used in processing text files. Filtering text file data can be made much easier with utilities such as `grep`, `egrep`, `fgrep`, and `cut`. Once that data is filtered, you may want to format it for viewing using `sort`, `nl`, or even the `cat` utility. If you need some statistical information on your text file, such as the number of lines it contains, the `wc` command is handy.

Explain both the structures and commands for redirection. Employing STDOUT, STDERR, and STDIN redirection allows rather complex filtering and processing of text. The `echo` command can assist in this process. You can also use pipelines of commands to perform redirection and produce excellent data for review. In addition, pipelines can be used in creating commands on the fly with utilities, such as `xargs`.

Review Questions

You can find the answers in the appendix.

1. On Linux systems, which file typically now points to a shell program instead of holding a shell program?
   - **A.** `/bin/bash`
   - **B.** `/bin/dash`
   - **C.** `/bin/zsh`
   - **D.** `/bin/sh`
   - **E.** `/bin/tcsh`
2. To see only the current Linux kernel version, which command should you use?
   - **A.** `uname`
   - **B.** `echo $BASH_VERSION`
   - **C.** `uname -r`
   - **D.** `uname -a`
   - **E.** `echo $SHELL`
3. What will the `echo \^New \^Style` command display?
   - **A.** `\^New \^Style`
   - **B.** `New Style`
   - **C.** `Style New`
   - **D.** `^New ^Style`
   - **E.** `\ew \tyle`
4. You need to determine if the `fortytwo.sh` program is in a `$PATH` directory. Which of the following commands will assist you in this task? (Choose all that apply.)
   - **A.** `which fortytwo.sh`
   - **B.** `cat fortytwo.sh`
   - **C.** `echo $PATH`
   - **D.** `fortytwo.sh`
   - **E.** `/usr/bin/fortytwo.sh`
5. You want to edit the file `SpaceOpera.txt` and decide to use the `vim` editor to complete this task. Which of the following are `vim` modes you might employ? (Choose all that apply.)
   - **A.** Insert
   - **B.** Change
   - **C.** Command
   - **D.** Ex
   - **E.** Edit

6. You have a lengthy file named `FileA.txt`. What will the `head -15 FileA.txt` command do?
   - **A.** Display all but the last 15 lines of the file
   - **B.** Display all but the first 15 lines of the file
   - **C.** Display the first 15 lines of the file
   - **D.** Display the last 15 lines of the file
   - **E.** Generate an error message

7. You are trying to peruse a rather large text file. A co-worker suggests you use a pager. Which of the following best describes what your co-worker is recommending?
   - **A.** Use a utility that allows you to view the first few lines of the file.
   - **B.** Use a utility that allows you to view one text page at time.
   - **C.** Use a utility that allows you to search through the file.
   - **D.** Use a utility that allows you to filter out text in the file.
   - **E.** Use a utility that allows you to view the last few lines of the file.

8. Which of the following does not describe the `less` utility?
   - **A.** It does not read the entire file prior to displaying the file's first page.
   - **B.** You can use the up and down arrow keys to move through the file.
   - **C.** You press the spacebar to move forward a page.
   - **D.** You can use the Esc+V key combination to move backward a page.
   - **E.** You can press the X key to exit from the utility.

9. The `cat -E MyFile.txt` command is entered and at the end of every line displayed is a `$`. What does this indicate?
   - **A.** The text file has been corrupted somehow.
   - **B.** The text file records end in the ASCII character NUL.
   - **C.** The text file records end in the ASCII character LF.
   - **D.** The text file records end in the ASCII character $.
   - **E.** The text file records contain a $ at their end.

10. The `cut` utility often needs delimiters to process text records. Which of the following best describes a delimiter?
    - **A.** One or more characters that designate the beginning of a line in a record
    - **B.** One or more characters that designate the end of a line in a record
    - **C.** One or more characters that designate the end of a text file to a command-line text processing utility
    - **D.** A single space or a colon (:) that creates a boundary between different data items in a record
    - **E.** One or more characters that create a boundary between different data items in a record

11. Which of the following utilities change text within a file? (Choose all that apply.)

- **A.** `cut`
- **B.** `sort`
- **C.** `vim`
- **D.** `nano`
- **E.** `sed`

12. A Unicode-encoded text file, `MyUCode.txt`, needs to be perused. Before you decide what utility to use in order to view the file's contents, you employ the `wc` command on it. This utility displays `2020 6786 11328` to STDOUT. What of the following is true? (Choose all that apply.)

- **A.** The file has 2,020 lines in it.
- **B.** The file has 2,020 characters in it.
- **C.** The file has 6,786 words in it.
- **D.** The file has 11,328 characters in it.
- **E.** The file has 11,328 lines in it.

13. The `grep` utility can employ regular expressions in its *`PATTERN`*. Which of the following best describes a regular expression?

- **A.** A series of characters you define for a utility, which uses the characters to match the same characters in text files
- **B.** ASCII characters, such as LF and NUL, that a utility uses to filter text
- **C.** Wildcard characters, such as `*` and `?`, that a utility uses to filter text
- **D.** A pattern template you define for a utility, which uses the pattern to filter text
- **E.** Quotation marks (single or double) used around characters to prevent unexpected results

14. Which of the following is a BRE pattern that could be used with the `grep` command? (Choose all that apply.)

- **A.** `Sp?ce`
- **B.** `"Space, the .*frontier"`
- **C.** `^Space`
- **D.** `(lasting | final)`
- **E.** `frontier$`

15. You need to search through a large text file and find any record that contains either `Luke` or `Laura` at the record's beginning. Also, the phrase `"Father is"` must be located somewhere in the record's middle. Which of the following is an ERE pattern that could be used with the `egrep` command to find this record?

- **A.** `"Luke$|Laura$.*Father is"`
- **B.** `"^Luke|^Laura.Father is"`

C. `"(^Luke|^Laura).Father is"`

D. `"(Luke$|Laura$).* Father is$"`

E. `"(^Luke|^Laura).*Father is.* "`

16. Which of the following best defines a file descriptor?

A. An environment variable, such as `$PS1`

B. A number that represents a process's open files

C. Another term for the file's name

D. A six character name that represents standard output

E. A symbol that indicates the file's classification

17. A file `data.txt` needs to be sorted numerically and its output saved to a new file `newdata.txt`. Which of the following commands can accomplish this task? (Choose all that apply.)

A. `sort -n -o newdata.txt data.txt`

B. `sort -n data.txt > newdata.txt`

C. `sort -n -o data.txt newdata.txt`

D. `sort -o newdata.txt data.txt`

E. `sort data.txt > newdata.txt`

18. By default, STDOUT goes to what item?

A. `/dev/tty`*n*, where *n* is a number

B. `/dev/null`

C. `>`

D. `/dev/tty`

E. `pwd`

19. Which of the following commands will display the file `SpaceOpera.txt` to output as well as a copy of it to the file `SciFi.txt`?

A. `cat SpaceOpera.txt | tee SciFi.txt`

B. `cat SpaceOpera.txt > SciFi.txt`

C. `cat SpaceOpera.txt 2> SciFi.txt`

D. `cat SpaceOpera.txt SciFi.txt`

E. `cat SpaceOpera.txt &> SciFi.txt`

20. Which of the following commands will put any generated error messages into the black hole?

A. `sort SpaceOpera.txt 2> BlackHole`

B. `sort SpaceOpera.txt &> BlackHole`

C. `sort SpaceOpera.txt > BlackHole`

D. `sort SpaceOpera.txt 2> /dev/null`

E. `sort SpaceOpera.txt > /dev/null`