Chapter 4

Searching and Analyzing Text

- ✓ **Objective 1.2: Given a scenario, manage files and directories**
- ✓ **Objective 3.1: Given a scenario, create simple shell scripts to automate common tasks**

Managing a Linux server involves many important steps and decisions based on data. Trying to gather the information you need in an agile and efficient manner is crucial. There are many Linux structures and tools that can help you uncover the knowledge you seek quickly.

In this chapter, we'll add more items to your Linux command-line tool belt. We'll cover filtering and formatting text and the basics of redirection, all the way to editing text. Commands and concepts in this chapter will be built upon and used in later chapters.

Processing Text Files

Once you have found or created a text file, you may need to process it in some way to extract needed information. Understanding how to filter and format text will assist you in this endeavor. In the following sections, we'll take a look at tools and methods that will aid you in processing text files.

Filtering Text

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, here are a few basics you should understand about the `cut` command:

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 4.1.

TABLE 4.1 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 few `cut` commands in action will help demonstrate its capabilities. Listing 4.1 shows a few `cut` utility examples.

Listing 4.1: 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
[...]
$
```

```
$ cut -c 1-5 /etc/passwd
root:
bin:x
[...]
$
```

In Listing 4.1, the `head` command is used to display 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 that the colon is encased in quotation marks to avoid unexpected results. The `-f` option is used to specify that only fields `1` (username) and `7` (shell) should be displayed.

The second example in Listing 4.1 uses the `-c` option. In this case, the `nlist` argument is set to `1-5`, so every record's first five characters are 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.

Another nice tool for filtering text is our old friend the `grep` command. The `grep` command is powerful in its use of regular expressions, which will really help with filtering text files. But before we cover those, peruse Table 4.2 for commonly used `grep` utility options.

TABLE 4.2 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.

Many commands use *regular expressions*. A regular expression is a pattern template you define for a utility, such as `grep`, which uses the pattern to filter text. 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]`, or a range of characters, such as `[A-z]`. 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 a dollar sign (`$`) symbol to them.

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 4.2 shows some examples.

Listing 4.2: 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 4.2, 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 4.2 are searching for instances of the word `root` within the password file. Notice that the first command displays two lines from the file. The second 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`.

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 4.3 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 4.3: 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
$
```

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.

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, 4th Edition* by Richard Blum and Christine Bresnahan (Wiley, 2021).

Using ERE patterns can be rather tricky. A few examples employing `grep` with EREs are helpful, such as the ones shown in Listing 4.4.

Listing 4.4: 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 that 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 Listing 4.4's second example, notice that the `egrep` command is employed. The `egrep` command is equivalent to using the `grep -E` command. The ERE pattern here 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 `grep` to filter out data from your text files.

Formatting Text

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

The `sort` utility sorts a file's data. Keep in mind 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, simply enter the `sort` command followed by the name of the file you wish to sort. Listing 4.5 shows an example of this.

Listing 4.5: 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 4.6.

Listing 4.6: Using the `sort -n` command

```
$ cat counts.txt
105
8
37
42
54
$
```

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

In Listing 4.6, notice that the file has different numbers listed in an unsorted order. The second example attempts to numerically order the file, using the `sort` command with no options. This yields incorrect results. However, the third example uses the `sort -n` command, which properly orders the file numerically.

There are several useful options for the `sort` command. Commonly used switches are shown in Table 4.3.

TABLE 4.3 The `sort` command's commonly used options

Short	Long	Description
`-c`	`--check`	Check if file is already sorted. Produces no output if file is sorted. If file is not sorted, it displays the file name, the line number, the keyword `disorder`, and the first unordered line's text.
`-f`	`--ignore-case`	Consider lowercase characters as uppercase characters when sorting.
`-k` *n1* `[,`*n2*`]`	`--key=`*n1* `[,`*n2*`]`	Sort the file using the data in the *n1* field. May optionally specify a second sort field by following *n1* with a comma and specifying *n2*. Field delimiters are spaces by default.
`-M`	`--month-sort`	Display text in month of the year order. Months must be listed as standard three-letter abbreviations, such as JAN, FEB, MAR, and so on.
`-n`	`--numeric-sort`	Display text in numerical order.
`-o` *file*	`--output=`*file*	Create a new sorted file named *file*.
`-r`	`--reverse`	Display text in reverse sort order.

The `sort` utility is handy for formatting a small text file to help you understand the data it contains. Another useful command for formatting small text files is one we've already touched on: the `cat` command.

The `cat` command's original purpose in life was to concatenate files for display. That is where it gets its name. However, it is typically used to display a single file. Listing 4.7 is an example of concatenating two files to display their text contents one after the other.

Listing 4.7: 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 files displayed in Listing 4.7 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 4.4 has a few commonly used switches.

TABLE 4.4 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 a text file is causing some sort of odd problem when you're processing it, you can quickly see if any nonprintable characters are embedded. Listing 4.8 contains an example of this method.

Listing 4.8: Using the `cat` command to display nonprintable characters

```
$ cat bell.txt

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

In Listing 4.8, the first `cat` command displays the file, and it appears to simply contain a blank line. However, by employing the `-v` option, 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.

Another handy set of utilities for formatting text are the `pr` and `printf` commands. The `pr` utility was covered in Chapter 3, "Managing Files, Directories, and Text," so let's explore the `printf` command. Its entire purpose in life is to format and display text data. It has the following basic syntax:

```
printf FORMAT [ARGUMENT]...
```

The basic idea is that you provide text formatting via *`FORMAT`* for the *`ARGUMENT`*. A simple example is shown in Listing 4.9.

Listing 4.9: Employing the `printf` command

```
$ printf "%s\n" "Hello World"
Hello World
$
```

In Listing 4.9, the `printf` command uses the `%s\n` as the formatting description. It is enclosed within quotation marks to prevent unexpected results. The `%s` tells `printf` to print the string of characters listed in the *`ARGUMENT`*, which in this example is `Hello World`. The `\n` portion of the *`FORMAT`* tells the `printf` command to print a newline character after printing the string. This allows the prompt to display on a new line, instead of at the displayed string's end.

While the `pr` utility can handle formatting entire text files, the `printf` command is geared toward formatting the output of a single text line. You must incorporate other commands and write a Bash shell script for it to process a whole text file with it.

The formatting characters for the `printf` command are not too difficult once you have reviewed them. A few common ones are listed in Table 4.5.

TABLE 4.5 The `printf` command's commonly used *FORMAT* settings

FORMAT	Description
`%c`	Display the first *ARGUMENT* character.
`%d`	Display the *ARGUMENT* as a decimal integer number.
`%f`	Display the *ARGUMENT* as a floating-point number.
`%s`	Display the *ARGUMENT* as a character string.
`\%`	Display a percentage sign.
`\"`	Display a double quotation mark.
`\\`	Display a backslash.
`\f`	Include a form feed character.
`\n`	Include a newline character.
`\r`	Include a carriage return.
`\t`	Include a horizontal tab.

In Listing 4.10 the `printf` command is used to print a floating-point number, which has three digits after the decimal point. Only two are desired, so the `%.2f` format is used.

Listing 4.10: Using the `printf` command to format a floating-point number

```
$ printf "%.2f\n" 98.532
98.53
$
```

Formatting text data can be useful in uncovering information. Be sure to play around with all these commands to get some worthwhile experience.

Determining Word Count

Besides formatting data, gathering statistics on various text files can also be helpful when you are managing a server. 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 4.11 shows an example.

Listing 4.11: 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 4.6.

TABLE 4.6 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, configuration file line length 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 accidentally merged two configuration file lines. An example is shown in Listing 4.12.

Listing 4.12: Using the wc command to check line length

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

In Listing 4.12, 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.

Redirecting Input and Output

When processing text and text files to help you to gather data, you may want to save that data. 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 4.13.

Listing 4.13: 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 4.14.

Listing 4.14: 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 4.14, 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 4.14 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 4.15 shows an example of using this redirection operator.

Listing 4.15: 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 4.15 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 4.16 shows an example of redirecting standard error.

Listing 4.16: 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 4.16 was issued to find any files within 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 4.16 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 the 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 4.17.

Listing 4.17: Using an 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 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 4.18.

Listing 4.18: Employing a 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 4.18, the file Grades.txt contains various integers separated by a space. The second command utilizes the tr utility to change those spaces 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 4.7.

A practical example of redirecting STDOUT and STDIN involves the diff utility, covered in Chapter 3. The diff utility allows you to discover any disparities between two text files and change the differing text file so that the two files are identical. It involves a few steps. The first ones are shown in Listing 4.19 along with extra explanatory commands.

Listing 4.19: Using diff with redirection operators

```
$ pr -mtl 15 numbers.txt random.txt
42                                  42
2A                                  Flat Land
52                                  Schrodinger's Cat
0010 1010                           0010 1010
*                                   0000 0010
$
$ cp numbers.txt n.txt
$
$ diff -e n.txt random.txt > switch.sh
$
```

TABLE 4.7 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.

In Listing 4.19, the `pr` utility displays the two files `numbers.txt` and `random.txt` side by side. You can see that differences exist between these two files. A new copy of the `numbers.txt` file is created, so any changes are only made to the new file, `n.txt`, in case anything goes wrong. The `diff` command uses the `-e` switch to create an `ed` script. This script will make the `n.txt` file the same as the `random.txt` file.

Prior to enacting the created script, a few additional items must be added to it. In Listing 4.20, the `echo` command is used two times to append letters to the script.

Listing 4.20: Update an `ed` script via redirection operators

```
$ echo w >> switch.sh
$ echo q >> switch.sh
$
$ cat switch.sh
5c
0000 0010
.
```

```
2,3c
Flat Land
Schrodinger's Cat
.
w
q
$
```

In Listing 4.20, the last command displays the `ed` script, `switch.sh`, to standard output. This script will modify the `n.txt` file, as shown in Listing 4.21.

Listing 4.21: Modifying a file via an `ed` script

```
$ diff -q n.txt random.txt
Files n.txt and random.txt differ
$
$ ed n.txt < switch.sh
21
52
$
$ diff -q n.txt random.txt
$
```

In Listing 4.21, the `diff` command does a simple comparison between the two files. Notice that it sends a message to STDOUT that the files are different. Then the `ed` utility is employed. To enact the script created by the `diff` command in Listing 4.21, the STDIN redirection operator is used. The last command in Listing 4.21 shows that there are now no differences between these two files.

Piping Commands

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:

```
command 1 | command 2 [| command n]...
```

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 4.22 shows a simple use of pipe redirection.

Listing 4.22: Employing pipe redirection

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

In Listing 4.22, 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 creative using pipe redirection. Listing 4.23 shows a command employing over four different utilities in a pipeline to audit accounts using the `/sbin/nologin` default shell.

Listing 4.23: Employing pipe redirection for several commands

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

In Listing 4.23, 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 4.24 contains an example of this handy command.

Listing 4.24: 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 4.24 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.

Creating Here Documents

Another form of STDIN redirection can be accomplished using a *here document*, which is sometimes called here text or heredoc. A here document allows you to redirect multiple items into a command. It can also modify a file using a script, create a script, keep data in a script, and so on.

A here document redirection operator is << followed by a keyword. This keyword can be anything, and it signals the beginning of the data as well as the data's end. Listing 4.25 shows an example of using the `sort` command along with a here document.

Listing 4.25: Employing a here document with the `sort` command

```
$ sort <<EOF
> dog
> cat
> fish
> EOF
cat
dog
fish
$
```

In Listing 4.25, the `sort` command is entered followed by the << redirection operator and a keyword, `EOF`. The Enter key is pressed, and a secondary prompt, >, appears, indicating that more data can be entered. Three words to be sorted are entered. The keyword,

EOF, is entered again to denote that data entry is complete. When this occurs, the sort utility alphabetically sorts the words and displays the results to STDOUT.

Creating Command Lines

Creating command-line commands is a useful skill. There are several different methods you can use. One such method is using 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 4.26 shows an example of doing this.

Listing 4.26: Employing the xargs command

```
$ find tmp -size 0
tmp/EmptyFile1.txt
tmp/EmptyFile2.txt
tmp/EmptyFile3.txt
$
$ find tmp -size 0 | xargs /usr/bin/ls
tmp/EmptyFile1.txt  tmp/EmptyFile2.txt  tmp/EmptyFile3.txt
$
```

In Listing 4.26, the first command finds any files in the tmp subdirectory that are empty (-size 0). The second command does the same thing, except this time, the output from the find command is piped as STDIN into the xargs utility. The xargs command uses the ls command to list the files. Notice that xargs requires not only the ls command's name but also its program's location in the virtual directory tree.

While Listing 4.26's commands are educational, they are not practical, because you get the same information just using the find utility. Listing 4.27 shows a functional use of employing the xargs utility.

Listing 4.27: Using the xargs command to delete files

```
$ find tmp -size 0 | xargs -p /usr/bin/rm
/usr/bin/rm tmp/EmptyFile1.txt tmp/EmptyFile2.txt tmp/EmptyFile3.txt
?...y
$
```

The xargs command used in Listing 4.27 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 created command is going to remove all three empty files with one rm command. After you type **y** and press the Enter key, the command is enacted, and the three files are deleted. This is a pretty handy way to find and remove unwanted files.

The other methods to create command-line commands on the fly use shell expansion. The first method puts a command to execute within parentheses and precedes it with a dollar sign. An example of this method is shown in Listing 4.28.

Listing 4.28: Using the `$()` method to create commands

```
$ touch tmp/EmptyFile1.txt
$ touch tmp/EmptyFile2.txt
$ touch tmp/EmptyFile3.txt
$
$ ls $(find tmp -size 0)
tmp/EmptyFile1.txt  tmp/EmptyFile2.txt  tmp/EmptyFile3.txt
$
```

In Listing 4.28, the `find` command is again used to locate any empty files in the `tmp` subdirectory. Because the command is encased by the `$()` symbols, it does not display to STDOUT. Instead, the filenames are passed to the `ls` utility, which does display the files to STDOUT. Of course, it would be more useful to delete those files, but they are needed in the next few examples.

The next method puts a command to execute within backticks (`). Be aware that backticks are not single quotation marks. You can typically find the backtick on the same keyboard key as the tilde (~) symbol. An example of this method is shown in Listing 4.29.

Listing 4.29: Using the backtick method to create commands

```
$ ls `find tmp -size 0`
tmp/EmptyFile1.txt  tmp/EmptyFile2.txt  tmp/EmptyFile3.txt
$
```

Notice in Listing 4.29 that the created command-line command behaves exactly as the constructed command in Listing 4.28. The command between the backticks executes and its output is passed as input to the `ls` utility.

Backticks are not very popular anymore. While they perform the same duty as do the `$()` symbols for creating commands, they are harder to see and are often confused with single quotation marks.

Another method for creating commands is brace expansion. This handy approach allows you to cut down on typing at the command line. Listing 4.30 provides a useful example of brace expansion.

Listing 4.30: Using brace expansion to create commands

```
$ rm -i tmp/EmptyFile{1,3}.txt
rm: remove regular empty file 'tmp/EmptyFile1.txt'? y
rm: remove regular empty file 'tmp/EmptyFile3.txt'? y
$
```

Notice in Listing 4.30 that two files are deleted. Instead of typing out the entire filenames, you can employ curly braces ({}). These curly braces contain two numbers separated by

a comma. This causes the `rm` utility to substitute a `1` in the braces' location for the first filename and a `3` for the second file's name. In essence, the brace expansion method allows you to denote multiple substitutions within a command argument. Thus, you can get very creative when building commands on the fly.

Editing Text Files

Manipulating text is performed on a regular basis when managing a Linux system. You may need to employ either a stream editor or a full-fledged interactive text editor to accomplish the task. In the following sections, we'll cover both types of editors.

Appreciating Text Editors

Whether you need to modify a configuration file or create a shell script, being able to use an interactive text file editor is an important skill. Also, it is a good idea to know more than just one. Therefore, we'll cover both the nano and the `vim` text editors.

The nano editor is a good text editor to start using if you have never dealt with an editor or have only used GUI editors. To start using the nano text editor, type **nano** followed by the name of the file you wish to edit or create. Figure 4.1 shows a nano text editor screen in action.

FIGURE 4.1 Using the nano text editor

In Figure 4.1 you can see the four main sections of the nano text editor. They are as follows:

Title Bar The title bar is at the nano text editor window's top line. It shows the current editor version as well as the name of the file you are presently editing. In Figure 4.1, the file being edited is the `test1.txt` file. If you simply typed **nano** and did not include a filename, you would see `New Buffer` in the title bar.

Main Body The nano text editor's main body is where you perform the editing. If the file already contains text, its first lines are shown in this area. If you need to view text that is not in the main body, you can use either arrow keys, the Page Up or Page Down key, and/or the page movement shortcut key combinations to move through the text.

Status Bar The status bar does not always contain information. It only displays status information for certain events. For example, in Figure 4.1, the text file has just been opened in nano, so the status bar area displays `[ Read 3 lines ]` to indicate that three text lines were read into the editor.

Shortcut List The shortcut list is one of the editor's most useful features. By glancing at this list at the window's bottom, you can see 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 remove text from the file, you highlight the text first, then press and hold the Ctrl key and then press the K 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, the `vim` editor is a popular choice.

Before we start talking about how to use 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 that 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 4.31 shows using the `which` utility to determine what command a Red Hat Linux distribution is using.

Listing 4.31: Using `which` to determine which command

```
$ which vim
/usr/bin/vim
$
```

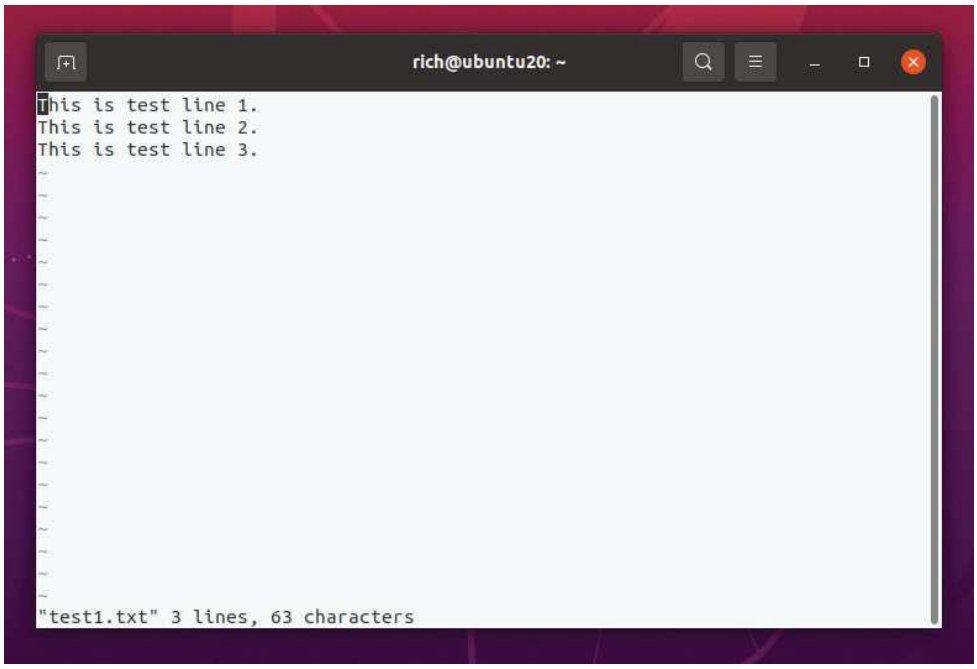
```
$ which vi
alias vi='vim'
        /usr/bin/vim
$
```

Listing 4.31 shows that this Red Hat Linux 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 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`**. Once 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 13, "Governing Software").

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 4.2 shows a `vim` text editor screen in action.

FIGURE 4.2 Using the `vim` text editor



In Figure 4.2 the file being edited is the `test1.txt` file again. The `vim` editor loads 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.

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

Command Mode This is the mode `vim` uses when you first enter the buffer area; this 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. This 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 by a colon (`:`). For example, to leave the `vim` editor and not save any changes you type **`:q`** and press the Enter key.

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 4.8 contains several moving commands for your perusal.

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. For example, when you move your cursor to a word's first letter and press CW, the word is deleted, and you are thrown into insert mode. You can then type the new word and press Esc to leave insert mode.

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.

Once you have made any needed text changes in the `vim` buffer area, it's time to save your work. You can type **`ZZ`** in command mode to write the buffer to disk and exit your process from the `vim` editor.

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 4.8 Commonly used `vim` command mode moving commands

Keystroke	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).
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.

Table 4.9 shows several Ex commands that can help you manage your text file. Notice that all the keystrokes include the necessary colon (`:`) to use Ex commands.

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.

TABLE 4.9 Commonly used `vim` Ex mode commands

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

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 in the Ex mode command **`:q`** to quit.

It's tempting to learn only one text editor and ignore the other. This, of course, won't help you pass the CompTIA Linux+ certification exam. But, in addition, 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` editor is king. Both are worth your time to master.

Learning about Stream Editors

There are times where you will want to edit text files without having to pull out a full-fledged text editor. In these cases, learning about two very popular *stream editors* is worthwhile. A stream editor modifies text that is passed to it via a file or output from a pipeline. The editor uses special commands to make text changes as the text "streams" through the editor utility.

The first stream editor we'll explore is called the *stream editor*. The command to invoke it 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 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:

- Reads one text line at a time from the input stream
- Matches that text with the supplied editor commands
- Modifies the text as specified in the commands
- Outputs the modified text to STDOUT

After the `sed` editor matches all the prespecified 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 according to the prespecified commands. An example is shown in Listing 4.32.

Listing 4.32: Using `sed` to modify STDIN text

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

Notice in Listing 4.32 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 by 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 4.33 shows an example of this.

Listing 4.33: 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 4.33, 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 4.34 shows an example of this.

Listing 4.34: 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 4.34, 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.

The stream editor has some rather useful command options. Commonly used ones are displayed in Table 4.10.

TABLE 4.10 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 4.35.

Listing 4.35: 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 4.35. 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.

If you have a lot of `sed` script commands, you can store them in a file. This is convenient because you can use the script file over and over again. Listing 4.36 shows an example of using a `sed` script one time.

Listing 4.36: Using `sed -f` to use a script file

```
$ cat script.sed
s/cake/donuts/
s/like/love/
$
$ sed -f script.sed cake.txt
Christine loves chocolate donuts.
Rich loves lemon donuts.
Tim only loves yellow donuts.
Samantha does not love donuts.
$
```

In Listing 4.36, notice that the `sed` script has single `sed` commands on each file line. No single quotation marks are employed here. Once the `sed` command is used along with the `-f` option and script file argument, the changes are applied to the file data and displayed STDOUT.

The `gawk` utility is also a stream editor, but it provides a more powerful editing process through its programming language. With the `gawk` programming language, you can do the following:

- Define variables to store data.
- Use arithmetic and string operators to work on data.
- Use programming structures, such as loops, to add logic to your processing.
- Create formatted reports from data.

The gawk programming language is popular for creating formatted reports from large datasets. You can create gawk programs and store them in files for repeated use.

A little confusion exists between awk and gawk, so let's address that before delving further into the gawk utility. The awk program was created for the Unix operating system, so when the GNU project rewrote it, they named it GNU awk, or gawk for short. However, on many distributions you can use either command, but they both actually call the gawk program. Listing 4.37 shows an example of this on an Ubuntu distribution.

Listing 4.37: Looking at the awk and gawk commands

```
$ which awk
/usr/bin/awk
$
$ readlink -f /usr/bin/awk
/usr/bin/gawk
$
$ which gawk
/usr/bin/gawk
$
```

In Listing 4.37, you can see that the awk command exists on this distribution. However, when you follow the soft link trail to the actual program used, it points to the gawk program. The gawk command exists as well.

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

```
gawk  [OPTIONS] [PROGRAM]... [FILENAME]
```

Similar to sed, you can provide the program on the same command line as the gawk command. It also employs the use of single quotation marks to enclose the script. However, unlike sed, the gawk utility requires you to put your programming language commands between two curly braces. An example is shown in Listing 4.38.

Listing 4.38: Using gawk to modify STDIN text

```
$ echo "Hello World" | gawk '{print $0}'
Hello World
$
$ echo "Hello World" | gawk '{print $1}'
Hello
$
$ echo "Hello World" | gawk '{print $2}'
```

```
World
$
```

The `print` command displays text to STDOUT, but notice that different parts of STDIN are shown, as you can see in Listing 4.38. This is accomplished through the `gawk` utility's defined data field variables. They are defined as follows:

- The `$0` variable represents the entire text line.
- The `$1` variable represents the text line's first data field.
- The `$2` variable represents the text line's second data field.
- The `$`*n* variable represents the text line's *n*th data field.

The `gawk` utility can also process text data from a file. An example of this is shown in Listing 4.39.

Listing 4.39: Using gawk 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.
$
$ gawk '{print $1}' cake.txt
Christine
Rich
Tim
Samantha
$
```

The `gawk` programming language is rather powerful and allows you to use many typical structures employed in other programming languages. In Listing 4.40, an attempt is made to change the word `cake` in the output to `donut`.

Listing 4.40: Using gawk structured commands to modify file text

```
$ gawk '{$4="donuts"; print $0}' cake.txt
Christine likes chocolate donuts
Rich likes lemon donuts
Tim only likes donuts cake.
Samantha does not donuts cake.
$
```

```
$ gawk '{if ($4 == "cake.") {$4="donuts"; print $0}}' cake.txt
Christine likes chocolate donuts
Rich likes lemon donuts
$
```

In Listing 4.40, the first attempt to substitute the words does not work properly. That is a result of two text file lines having the word `cake` in data field `$5` instead of data field `$4`. The second `gawk` attempt employs an `if` statement to check if data field `$4` is equal to the word `cake`. If the statement returns true, the data field is changed to `donuts` and the text line is displayed on STDOUT. Otherwise, the text line is ignored.

Using complex programming structures in `gawk` can be tricky on the command line. It's much better to put those commands in a file. However, you need to know a few common `gawk` options prior to doing this. Table 4.11 has some typical `gawk` switches.

TABLE 4.11 The gawk command's commonly used options

Short	Long	Description
`-F` *d*	`--field-separator`*d*	Specify the delimiter that separates the data file's fields.
`-f` *file*	`--file=`*file*	Use program in *file* for text processing.
`-s`	`--sandbox`	Execute gawk program in sandbox mode.

Using the field separator option is very handy when the data file's fields are separated by commas or colons. An example of pulling data from the password file using this switch is shown in Listing 4.41.

Listing 4.41: Using gawk structured commands to extact file data

```
$ gawk -F : '{print $1}' /etc/passwd
root
bin
daemon
[...]
$
```

You can put complex `gawk` programs into files to keep them for reuse. Listing 4.42 shows an example of this.

Listing 4.42: Using a gawk program file

```
$ cat cake.txt
Christine likes chocolate cake.
Rich likes lemon cake.
Tim only likes yellow cake.
Samantha does not like cake.
$
$ cat script.gawk
{if ($4=="cake.")
  {$4="donuts"; print $0}
else if ($5=="cake.")
  {$5="donuts"; print $0}}
$
$ gawk -f script.gawk cake.txt
Christine likes chocolate donuts
Rich likes lemon donuts
Tim only likes yellow donuts
Samantha does not like donuts
$
```

In Listing 4.42, a more complex `if` structure statement is written using the `gawk` programming language and saved to a file, `script.gawk`. This script not only employs an `if` statement, it also incorporates an `else if` structure. Notice also that no single quotation marks are needed when the `gawk` program is stored in a file. Using the `-f` switch, the program is enacted on the `cake.txt` file, and the appropriate word is changed in every line.

Summary

Being able to process data to make agile decisions is important for administering your Linux system. There are many Linux structures and tools, which can help you in uncovering the information you need.

This chapter's purpose was to continue to improve your Linux command-line tool belt. The tools and structures added in this chapter will allow you to search and analyze text in order to uncover knowledge in an efficient manner.

Exam Essentials

Summarize the various utilities used in processing text files. Filtering text file data can be made much easier with utilities such as `grep`, `egrep`, and `cut`. Once that data is filtered, you may want to format it for viewing using `sort`, `pr`, `printf`, 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 as well as here documents. You can also use pipelines of commands to perform redirection and produce excellent data for review.

Describe the various methods used for editing text files. 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 `sed` and its scripts or the `gawk` programming language.

Review Questions

1. 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.

2. 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

3. Which of the following utilities change text within a file? (Choose all that apply.)
   - **A.** `cut`
   - **B.** `sort`
   - **C.** `vim`
   - **D.** `nano`
   - **E.** `sed`

4. You have a text file, `monitor.txt`, which contains information concerning the monitors used within the data center. Each record ends with the ASCII LF character and fields are delimitated by a comma (,). A text record has the monitor ID, manufacture, serial number, and location. To display each data center monitor's monitor ID, serial number, and location, you'd use which `cut` command?
   - **A.** `cut -d "," -f 1,3,4 monitor.txt`
   - **B.** `cut -z -d "," -f 1,3,4 monitor.txt`
   - **C.** `cut -f "," -d 1,3,4 monitor.txt`
   - **D.** `cut monitor.txt -d "," -f 1,3,4`
   - **E.** `cut monitor.txt -f "," -d 1,3,4`

5. 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

6. You are a system administrator on a Red Hat Linux server. You need to view records in the `/var/log/messages` file that start with the date May 30 and end with the IPv4 address 192.168.10.42. Which of the following is the best `grep` command to use?
   - **A.** `grep "May 30?192.168.10.42" /var/log/messages`
   - **B.** `grep "May 30.*192.168.10.42" /var/log/messages`
   - **C.** `grep -i "May 30.*192.168.10.42" /var/log/messages`
   - **D.** `grep -i "May 30?192.168.10.42" /var/log/messages`
   - **E.** `grep -v "May 30.*192.168.10.42" /var/log/messages`

7. 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$`

8. 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.*"`

9. 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`

10. Which of the following commands can display the `data.txt` and `datatoo.txt` files' content one after the other to STDOUT? (Choose all that apply.)
   A. `ls data.txt datatoo.txt`
   B. `sort -n data.txt > datatoo.txt`
   C. `cat -n data.txt datatoo.txt`
   D. `ls -l data.txt datatoo.txt`
   E. `sort data.txt datatoo.txt`

11. A text file, `StarGateAttacks.txt`, needs to be specially formatted for review. Which of the following commands is the best command to accomplish this task quickly?
   A. `printf`
   B. `wc`
   C. `pr`
   D. `paste`
   E. `nano`

12. You need to format the string `42.777` into the correct two-digit floating number. Which of the following `printf` command *`FORMAT`* settings is the correct one to use?
   A. `"%s\n"`
   B. `"%.2s\n"`
   C. `"%d\n"`
   D. `"%.2c\n"`
   E. `"%.2f\n"`

13. A Unicode-encoded text file, `MyUCode.txt`, needs to be perused. Before you decide what utility to use in order view the file's contents, you employ the `wc` command on it. This utility displays `2020 6786 11328` to STDOUT. Which 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.

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

A. A letter that represents the file's type

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

15. 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`

16. Which of the following commands will display the file `SpaceOpera.txt` to output as well as save 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. `cp SpaceOpera.txt SciFi.txt`

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

17. 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`

18. Which of the following commands will determine how many records in the file `Problems.txt` contain the word `error`?

A. `grep error Problems.txt | wc -b`

B. `grep error Problems.txt | wc -w`

C. `grep error Problems.txt | wc -l`

D. `grep Problems.txt error | wc -w`

E. `grep Problems.txt error | wc -l`

19. You want to find any file named `42.tmp`, which exists somewhere in your current directory's tree structure and display its contents to STDOUT. Which of the following will allow you to build a command to do this? (Choose all that apply.)

    A. `xargs (find . -name 42.tmp) cat`

    B. `cat `find . -name 42.tmp``

    C. `cat $(find . -name 42.tmp)`

    D. `cat {find . -name 42.tmp}`

    E. `find . -name 42.tmp | xargs cat`

20. 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