

Presenting Data

IN THIS CHAPTER

Revisiting redirection

Standard input and output

Reporting errors

Throwing away data

Creating log files

So far the scripts shown in this book display information either by echoing data to the monitor or by redirecting data to a file. Chapter 11 demonstrated how to redirect the output of a command to a file. This chapter expands on that topic by showing you how you can redirect the output of your script to different locations on your Linux system.

Understanding Input and Output

So far, you've seen two methods for displaying the output from your scripts:

- Displaying output on the monitor screen
- Redirecting output to a file

Both methods produced an all-or-nothing approach to data output. There are times, however, when it would be nice to display some data on the monitor and other data in a file. For these instances, it comes in handy to know how Linux handles input and output so you can get your script output to the right place.

The following sections describe how to use the standard Linux input and output system to your advantage, to help direct script output to specific locations.

Standard file descriptors

The Linux system handles every object as a file. This includes the input and output process. Linux identifies each file object using a *file descriptor*. The file descriptor is a non-negative integer that

uniquely identifies open files in a session. Each process is allowed to have up to nine open file descriptors at a time. The bash shell reserves the first three file descriptors (0, 1, and 2) for special purposes. These are shown in Table 15-1.

TABLE 15-1 Linux Standard File Descriptors

File Descriptor	Abbreviation	Description
0	STDIN	Standard input
1	STDOUT	Standard output
2	STDERR	Standard error

These three special file descriptors handle the input and output from your script. The shell uses them to direct the default input and output in the shell to the appropriate location, which by default is usually your monitor. The following sections describe each of these standard file descriptors in greater detail.

STDIN

The `STDIN` file descriptor references the standard input to the shell. For a terminal interface, the standard input is the keyboard. The shell receives input from the keyboard on the `STDIN` file descriptor and processes each character as you type it.

When you use the input redirect symbol (`<`), Linux replaces the standard input file descriptor with the file referenced by the redirection. It reads the file and retrieves data just as if it were typed on the keyboard.

Many bash commands accept input from `STDIN`, especially if no files are specified on the command line. Here's an example of using the `cat` command with data entered from `STDIN`:

```
$ cat
this is a test
this is a test
this is a second test.
this is a second test.
```

When you enter the `cat` command on the command line by itself, it accepts input from `STDIN`. As you enter each line, the `cat` command echoes the line to the display.

However, you can also use the `STDIN` redirect symbol to force the `cat` command to accept input from another file other than `STDIN`:

```
$ cat < testfile
This is the first line.
This is the second line.
```

```
This is the third line.
$
```

Now the `cat` command uses the lines that are contained in the `testfile` file as the input. You can use this technique to input data to any shell command that accepts data from `STDIN`.

STDOUT

The `STDOUT` file descriptor references the standard output for the shell. On a terminal interface, the standard output is the terminal monitor. All output from the shell (including programs and scripts you run in the shell) is directed to the standard output, which is the monitor.

Most bash commands direct their output to the `STDOUT` file descriptor by default. As shown in Chapter 11, you can change that using output redirection:

```
$ ls -l > test2
$ cat test2
total 20
-rw-rw-r-- 1 rich rich 53 2014-10-16 11:30 test
-rw-rw-r-- 1 rich rich  0 2014-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 73 2014-10-16 11:23 testfile
$
```

With the output redirection symbol, all the output that normally would go to the monitor is instead redirected to the designated redirection file by the shell.

You can also append data to a file. You do this using the `>>` symbol:

```
$ who >> test2
$ cat test2
total 20
-rw-rw-r-- 1 rich rich 53 2014-10-16 11:30 test
-rw-rw-r-- 1 rich rich  0 2014-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 73 2014-10-16 11:23 testfile
rich      pts/0          2014-10-17 15:34 (192.168.1.2)
$
```

The output generated by the `who` command is appended to the data already in the `test2` file.

However, if you use the standard output redirection for your scripts, you can run into a problem. Here's an example of what can happen in your script:

```
$ ls -al badfile > test3
ls: cannot access badfile: No such file or directory
$ cat test3
$
```

When a command produces an error message, the shell doesn't redirect the error message to the output redirection file. The shell created the output redirection file, but the error message appeared on the monitor screen. Notice that there isn't an error when trying to display the contents of the `test3` file. The `test3` file was created just fine, but it's empty.

The shell handles error messages separately from the normal output. If you're creating a shell script that runs in background mode, often you must rely on the output messages being sent to a log file. Using this technique, if any error messages occur, they don't appear in the log file. You need to do something different.

STDERR

The shell handles error messages using the special `STDERR` file descriptor. The `STDERR` file descriptor references the standard error output for the shell. This is the location where the shell sends error messages generated by the shell or programs and scripts running in the shell.

By default, the `STDERR` file descriptor points to the same place as the `STDOUT` file descriptor (even though they are assigned different file descriptor values). This means that, by default, all error messages go to the monitor display.

However, as you saw in the example, when you redirect `STDOUT`, this doesn't automatically redirect `STDERR`. When working with scripts, you'll often want to change that behavior, especially if you're interested in logging error messages to a log file.

Redirecting errors

You've already seen how to redirect the `STDOUT` data by using the redirection symbol. Redirecting the `STDERR` data isn't much different; you just need to define the `STDERR` file descriptor when you use the redirection symbol. You can do this in a couple of ways.

Redirecting errors only

As you saw in Table 15-1, the `STDERR` file descriptor is set to the value 2. You can select to redirect only error messages by placing this file descriptor value immediately before the redirection symbol. The value must appear immediately before the redirection symbol or it doesn't work:

```
$ ls -al badfile 2> test4
$ cat test4
ls: cannot access badfile: No such file or directory
$
```

Now when you run the command, the error message doesn't appear on the monitor. Instead, the output file contains any error messages that are generated by the command. Using this method, the shell redirects the error messages only, not the normal data. Here's another example of mixing `STDOUT` and `STDERR` messages in the same output:

```
$ ls -al test badtest test2 2> test5
-rw-rw-r-- 1 rich rich 158 2014-10-16 11:32 test2
```

```
$ cat test5
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$
```

The normal `STDOUT` output from the `ls` command still goes to the default `STDOUT` file descriptor, which is the monitor. Because the command redirects file descriptor 2 output (`STDERR`) to an output file, the shell sends any error messages generated directly to the specified redirection file.

Redirecting errors and data

If you want to redirect both errors and the normal output, you need to use two redirection symbols. You need to precede each with the appropriate file descriptor for the data you want to redirect and then have them point to the appropriate output file for holding the data:

```
$ ls -al test test2 test3 badtest 2> test6 1> test7
$ cat test6
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
$ cat test7
-rw-rw-r-- 1 rich rich 158 2014-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 0 2014-10-16 11:33 test3
$
```

The shell redirects the normal output of the `ls` command that would have gone to `STDOUT` to the `test7` file using the `1>` symbol. Any error messages that would have gone to `STDERR` were redirected to the `test6` file using the `2>` symbol.

You can use this technique to separate normal script output from any error messages that occur in the script. This allows you to easily identify errors without having to wade through thousands of lines of normal output data.

Alternatively, if you want, you can redirect both `STDERR` and `STDOUT` output to the same output file. The bash shell provides a special redirection symbol just for this purpose, the `&>` symbol:

```
$ ls -al test test2 test3 badtest &> test7
$ cat test7
ls: cannot access test: No such file or directory
ls: cannot access badtest: No such file or directory
-rw-rw-r-- 1 rich rich 158 2014-10-16 11:32 test2
-rw-rw-r-- 1 rich rich 0 2014-10-16 11:33 test3
$
```

When you use the `&>` symbol, all the output generated by the command is sent to the same location, both data and errors. Notice that one of the error messages is out of order from what you'd expect. The error message for the `badtest` file (the last file to be listed)

appears second in the output file. The bash shell automatically gives error messages a higher priority than the standard output. This allows you to view the error messages together, rather than scattered throughout the output file.

Redirecting Output in Scripts

You can use the `STDOUT` and `STDERR` file descriptors in your scripts to produce output in multiple locations simply by redirecting the appropriate file descriptors. There are two methods for redirecting output in the script:

- Temporarily redirecting each line
- Permanently redirecting all commands in the script

The following sections describe how each of these methods works.

Temporary redirections

If you want to purposely generate error messages in your script, you can redirect an individual output line to `STDERR`. You just need to use the output redirection symbol to redirect the output to the `STDERR` file descriptor. When you redirect to a file descriptor, you must precede the file descriptor number with an ampersand (`&`):

```
echo "This is an error message" >&2
```

This line displays the text wherever the `STDERR` file descriptor for the script is pointing, instead of the normal `STDOUT`. The following is an example of a script that uses this feature:

```
$ cat test8
#!/bin/bash
# testing STDERR messages

echo "This is an error" >&2
echo "This is normal output"
$
```

If you run the script as normal, you don't notice any difference:

```
$ ./test8
This is an error
This is normal output
$
```

Remember that, by default, Linux directs the `STDERR` output to `STDOUT`. However, if you redirect `STDERR` when running the script, any text directed to `STDERR` in the script is redirected:

```
$ ./test8 2> test9
This is normal output
```

```
$ cat test9
This is an error
$
```

Perfect! The text displayed using `STDOUT` appears on the monitor, while the `echo` statement text sent to `STDERR` is redirected to the output file.

This method is great for generating error messages in your scripts. If someone uses your scripts, they can easily redirect the error messages using the `STDERR` file descriptor, as shown.

Permanent redirections

If you have lots of data that you're redirecting in your script, it can get tedious having to redirect every `echo` statement. Instead, you can tell the shell to redirect a specific file descriptor for the duration of the script by using the `exec` command:

```
$ cat test10
#!/bin/bash
# redirecting all output to a file
exec 1>testout

echo "This is a test of redirecting all output"
echo "from a script to another file."
echo "without having to redirect every individual line"
$ ./test10
$ cat testout
This is a test of redirecting all output
from a script to another file.
without having to redirect every individual line
$
```

The `exec` command starts a new shell and redirects the `STDOUT` file descriptor to a file. All output in the script that goes to `STDOUT` is instead redirected to the file.

You can also redirect the `STDOUT` in the middle of a script:

```
$ cat test11
#!/bin/bash
# redirecting output to different locations

exec 2>testerror

echo "This is the start of the script"
echo "now redirecting all output to another location"

exec 1>testout

echo "This output should go to the testout file"
```

```
echo "but this should go to the testerror file" >&2
$
$ ./test11
This is the start of the script
now redirecting all output to another location
$ cat testout
This output should go to the testout file
$ cat testerror
but this should go to the testerror file
$
```

The script uses the `exec` command to redirect any output going to `STDERR` to the file `testerror`. Next, the script uses the `echo` statement to display a few lines to `STDOUT`. After that, the `exec` command is used again to redirect `STDOUT` to the `testout` file. Notice that even when `STDOUT` is redirected, you can still specify the output from an `echo` statement to go to `STDERR`, which in this case is still redirected to the `testerror` file.

This feature can come in handy when you want to redirect the output of just parts of a script to an alternative location, such as an error log. There's just one problem you run into when using this.

After you redirect `STDOUT` or `STDERR`, you can't easily redirect them back to their original location. If you need to switch back and forth with your redirection, you need to learn a trick. The "Creating Your Own Redirection" section later in this chapter discusses this trick and how to use it in your shell scripts.

Redirecting Input in Scripts

You can use the same technique used to redirect `STDOUT` and `STDERR` in your scripts to redirect `STDIN` from the keyboard. The `exec` command allows you to redirect `STDIN` from a file on the Linux system:

```
exec 0< testfile
```

This command informs the shell that it should retrieve input from the file `testfile` instead of `STDIN`. This redirection applies anytime the script requests input. Here's an example of this in action:

```
$ cat test12
#!/bin/bash
# redirecting file input

exec 0< testfile
count=1

while read line
```



```

do
    echo "Line #${count}: $line"
    count=$((count + 1))
done
$ ./test12
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
$

```

Chapter 14 showed you how to use the `read` command to read data entered from the keyboard by a user. By redirecting `STDIN` from a file, when the `read` command attempts to read from `STDIN`, it retrieves data from the file instead of the keyboard.

This is an excellent technique to read data in files for processing in your scripts. A common task for Linux system administrators is to read data from log files for processing. This is the easiest way to accomplish that task.

Creating Your Own Redirection

When you redirect input and output in your script, you're not limited to the three default file descriptors. I mentioned that you could have up to nine open file descriptors in the shell. The other six file descriptors are numbered from 3 through 8 and are available for you to use as either input or output redirection. You can assign any of these file descriptors to a file and then use them in your scripts as well. This section shows you how to use the other file descriptors in your scripts.

Creating output file descriptors

You assign a file descriptor for output by using the `exec` command. As with the standard file descriptors, after you assign an alternative file descriptor to a file location, that redirection stays permanent until you reassign it. Here's a simple example of using an alternative file descriptor in a script:

```

$ cat test13
#!/bin/bash
# using an alternative file descriptor

exec 3>test13out

echo "This should display on the monitor"
echo "and this should be stored in the file" >&3
echo "Then this should be back on the monitor"
$ ./test13
This should display on the monitor

```

```
Then this should be back on the monitor
$ cat test13out
and this should be stored in the file
$
```

The script uses the `exec` command to redirect file descriptor 3 to an alternative file location. When the script executes the `echo` statements, they display on `STDOUT` as you would expect. However, the `echo` statements that you redirect to file descriptor 3 go to the alternative file. This allows you to keep normal output for the monitor and redirect special information to files, such as log files.

You can also use the `exec` command to append data to an existing file instead of creating a new file:

```
exec 3>>test13out
```

Now the output is appended to the `test13out` file instead of creating a new file.

Redirecting file descriptors

Here's the trick to help you bring back a redirected file descriptor. You can assign an alternative file descriptor to a standard file descriptor, and vice versa. This means that you can redirect the original location of `STDOUT` to an alternative file descriptor and then redirect that file descriptor back to `STDOUT`. This might sound somewhat complicated, but in practice it's fairly straightforward. This example will clear things up for you:

```
$ cat test14
#!/bin/bash
# storing STDOUT, then coming back to it

exec 3>&1
exec 1>test14out

echo "This should store in the output file"
echo "along with this line."

exec 1>&3

echo "Now things should be back to normal"
$
$ ./test14
Now things should be back to normal
$ cat test14out
This should store in the output file
along with this line.
$
```

This example is a little crazy so let's walk through it piece by piece. First, the script redirects file descriptor 3 to the current location of file descriptor 1, which is `STDOUT`. This means that any output sent to file descriptor 3 goes to the monitor.

The second `exec` command redirects `STDOUT` to a file. The shell now redirects any output sent to `STDOUT` directly to the output file. However, file descriptor 3 still points to the original location of `STDOUT`, which is the monitor. If you send output data to file descriptor 3 at this point, it still goes to the monitor, even though `STDOUT` is redirected.

After sending some output to `STDOUT`, which points to a file, the script then redirects `STDOUT` to the current location of file descriptor 3, which is still set to the monitor. This means that now `STDOUT` points to its original location, the monitor.

This method can get confusing, but it's a common way to temporarily redirect output in script files and then set the output back to the normal settings.

Creating input file descriptors

You can redirect input file descriptors exactly the same way as output file descriptors. Save the `STDIN` file descriptor location to another file descriptor before redirecting it to a file; when you're finished reading the file, you can restore `STDIN` to its original location:

```
$ cat test15
#!/bin/bash
# redirecting input file descriptors

exec 6<&0

exec 0< testfile

count=1
while read line
do
    echo "Line #${count}: $line"
    count=$((count + 1))
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
Y|y) echo "Goodbye";;
N|n) echo "Sorry, this is the end.";;
esac
$ ./test15
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
Are you done now? y
Goodbye
$
```

In this example, file descriptor 6 is used to hold the location for `STDIN`. The script then redirects `STDIN` to a file. All the input for the `read` command comes from the redirected `STDIN`, which is now the input file.

When all the lines have been read, the script returns `STDIN` to its original location by redirecting it to file descriptor 6. The script tests to make sure that `STDIN` is back to normal by using another `read` command, which this time waits for input from the keyboard.

Creating a read/write file descriptor

As odd as it may seem, you can also open a single file descriptor for both input and output. You can then use the same file descriptor to both read data from a file and write data to the same file.

You need to be especially careful with this method, however. As you read and write data to and from a file, the shell maintains an internal pointer, indicating where it is in the file. Any reading or writing occurs where the file pointer last left off. This can produce some interesting results if you're not careful. Look at this example:

```
$ cat test16
#!/bin/bash
# testing input/output file descriptor

exec 3<> testfile
read line <&3
echo "Read: $line"
echo "This is a test line" >&3
$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ ./test16
Read: This is the first line.
$ cat testfile
This is the first line.
This is a test line
ine.
This is the third line.
$
```

This example uses the `exec` command to assign file descriptor 3 for both input and output sent to and from the file `testfile`. Next, it uses the `read` command to read the first line in the file, using the assigned file descriptor, and then it displays the read line of data in `STDOUT`. After that, it uses the `echo` statement to write a line of data to the file opened with the same file descriptor.

When you run the script, at first things look just fine. The output shows that the script read the first line in the `testfile` file. However, if you display the contents of the `testfile` file after running the script, you see that the data written to the file overwrote the existing data.

When the script writes data to the file, it starts where the file pointer is located. The `read` command reads the first line of data, so it left the file pointer pointing to the first character in the second line of data. When the `echo` statement outputs data to the file, it places the data at the current location of the file pointer, overwriting whatever data was there.

Closing file descriptors

If you create new input or output file descriptors, the shell automatically closes them when the script exits. There are situations, however, when you need to manually close a file descriptor before the end of the script.

To close a file descriptor, redirect it to the special symbol `&-`. This is how this looks in the script:

```
exec 3>&-
```

This statement closes file descriptor 3, preventing it from being used any more in the script. Here's an example of what happens when you try to use a closed file descriptor:

```
$ cat badtest
#!/bin/bash
# testing closing file descriptors

exec 3> test17file

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3
$ ./badtest
./badtest: 3: Bad file descriptor
$
```

After you close the file descriptor, you can't write any data to it in your script or the shell produces an error message.

There's yet another thing to be careful of when closing file descriptors. If you open the same output file later on in your script, the shell replaces the existing file with a new file. This means that if you output any data, it overwrites the existing file. Consider the following example of this problem:

```
$ cat test17
#!/bin/bash
# testing closing file descriptors

exec 3> test17file
echo "This is a test line of data" >&3
exec 3>&-

cat test17file

exec 3> test17file
echo "This'll be bad" >&3
$ ./test17
This is a test line of data
$ cat test17file
This'll be bad
$
```

After sending a data string to the `test17file` file and closing the file descriptor, the script uses the `cat` command to display the contents of the file. So far, so good. Next, the script reopens the output file and sends another data string to it. When you display the contents of the output file, all you see is the second data string. The shell overwrote the original output file.

Listing Open File Descriptors

With only nine file descriptors available to you, you'd think that it wouldn't be hard to keep things straight. Sometimes, however, it's easy to get lost when trying to keep track of which file descriptor is redirected where. To help you keep your sanity, the bash shell provides the `lsuf` command.

The `lsuf` command lists all the open file descriptors on the entire Linux system. This is somewhat of a controversial feature, because it can provide information about the Linux system to non-system-administrators. That's why many Linux systems hide this command so users don't accidentally stumble across it.

On many Linux systems (such as Fedora) the `lsuf` command is located in the `/usr/sbin` directory. To run it with a normal user account, I have to reference it by its full pathname:

```
$ /usr/sbin/lsuf
```

This produces an amazing amount of output. It displays information about every file currently open on the Linux system. This includes all the processes running on background, as well as any user accounts logged in to the system.

Plenty of command line parameters and options are available to help filter out the `lsuf` output. The most commonly used are `-p`, which allows you to specify a process ID (PID), and `-d`, which allows you to specify the file descriptor numbers to display.

To easily determine the current PID of the process, you can use the special environment variable `$$`, which the shell sets to the current PID. The `-a` option is used to perform a Boolean AND of the results of the other two options, to produce the following:

```
$ /usr/sbin/lsuf -a -p $$ -d 0,1,2
COMMAND PID USER  FD  TYPE DEVICE SIZE NODE NAME
bash    3344 rich   0u   CHR  136,0      2 /dev/pts/0
bash    3344 rich   1u   CHR  136,0      2 /dev/pts/0
bash    3344 rich   2u   CHR  136,0      2 /dev/pts/0
$
```

This shows the default file descriptors (0, 1, and 2) for the current process (the bash shell). The default output of `lsuf` contains several columns of information, described in Table 15-2.

TABLE 15-2 Default lsuf Output

Column	Description
COMMAND	The first nine characters of the name of the command in the process
PID	The process ID of the process
USER	The login name of the user who owns the process
FD	The file descriptor number and access type [r—(read), w—(write), u—(read/write)]
TYPE	The type of file [CHR—(character), BLK—(block), DIR—(directory), REG—(regular file)]
DEVICE	The device numbers (major and minor) of the device
SIZE	If available, the size of the file
NODE	The node number of the local file
NAME	The name of the file

The file type associated with `STDIN`, `STDOUT`, and `STDERR` is character mode. Because the `STDIN`, `STDOUT`, and `STDERR` file descriptors all point to the terminal, the name of the output file is the device name of the terminal. All three standard files are available for both reading and writing (although it does seem odd to be able to write to `STDIN` and read from `STDOUT`).

Now, let's look at the results of the `lsuf` command from inside a script that's opened a couple of alternative file descriptors:

```
$ cat test18
#!/bin/bash
# testing lsof with file descriptors

exec 3> test18file1
exec 6> test18file2
exec 7< testfile

/usr/sbin/lsof -a -p $$ -d0,1,2,3,6,7
$ ./test18
COMMAND  PID USER   FD   TYPE DEVICE SIZE  NODE NAME
test18   3594 rich    0u    CHR  136,0        2 /dev/pts/0
test18   3594 rich    1u    CHR  136,0        2 /dev/pts/0
test18   3594 rich    2u    CHR  136,0        2 /dev/pts/0
18  3594 rich    3w    REG  253,0        0 360712 /home/rich/test18file1
18  3594 rich    6w    REG  253,0        0 360715 /home/rich/test18file2
18  3594 rich    7r    REG  253,0       73 360717 /home/rich/testfile
$
```

The script creates three alternative file descriptors, two for output (3 and 6) and one for input (7). When the script runs the `lsof` command, you can see the new file descriptors in the output. We truncated the first part of the output so you could see the results of the filename. The filename shows the complete pathname for the files used in the file descriptors. It shows each of the files as type `REG`, which indicates that they are regular files on the filesystem.

Suppressing Command Output

Sometimes, you may not want to display any output from your script. This often occurs if you're running a script as a background process (see Chapter 16). If any error messages occur from the script while it's running in the background, the shell e-mails them to the owner of the process. This can get tedious, especially if you run scripts that generate minor nuisance errors.

To solve that problem, you can redirect `STDERR` to a special file called the *null file*. The null file is pretty much what it says it is — a file that contains nothing. Any data that the shell outputs to the null file is not saved, thus the data are lost.

The standard location for the null file on Linux systems is `/dev/null`. Any data you redirect to that location is thrown away and doesn't appear:

```
$ ls -al > /dev/null
$ cat /dev/null
$
```

This is a common way to suppress any error messages without actually saving them:


```
$ ls -al badfile test16 2> /dev/null
-rwxr--r--    1 rich    rich          135 Oct 29 19:57 test16*
$
```

You can also use the `/dev/null` file for input redirection as an input file. Because the `/dev/null` file contains nothing, it is often used by programmers to quickly remove data from an existing file without having to remove the file and re-create it:

```
$ cat testfile
This is the first line.
This is the second line.
This is the third line.
$ cat /dev/null > testfile
$ cat testfile
$
```

The file `testfile` still exists on the system, but now it is empty. This is a common method used to clear out log files that must remain in place for applications to operate.

Using Temporary Files

The Linux system contains a special directory location reserved for temporary files. Linux uses the `/tmp` directory for files that don't need to be kept indefinitely. Most Linux distributions configure the system to automatically remove any files in the `/tmp` directory at bootup.

Any user account on the system has privileges to read and write files in the `/tmp` directory. This feature provides an easy way for you to create temporary files that you don't necessarily have to worry about cleaning up.

There's even a specific command to use for creating a temporary file. The `mktemp` command allows you to easily create a unique temporary file in the `/tmp` folder. The shell creates the file but doesn't use your default `umask` value (see Chapter 7). Instead, it only assigns read and write permissions to the file's owner and makes you the owner of the file. After you create the file, you have full access to read and write to and from it from your script, but no one else can access it (other than the root user, of course).

Creating a local temporary file

By default, `mktemp` creates a file in the local directory. To create a temporary file in a local directory with the `mktemp` command, you just need to specify a filename template. The template consists of any text filename, plus six X's appended to the end of the filename:

```
$ mktemp testing.XXXXXX
$ ls -al testing*
-rw-----    1 rich    rich          0 Oct 17 21:30 testing.UfIi13
$
```

The `mktemp` command replaces the six X's with a six-character code to ensure the filename is unique in the directory. You can create multiple temporary files and be assured that each one is unique:

```
$ mktemp testing.XXXXXX
testing.1DRLuV
$ mktemp testing.XXXXXX
testing.lVBtkW
$ mktemp testing.XXXXXX
testing.PgqNKG
$ ls -l testing*
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.1DRLuV
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.PgqNKG
-rw----- 1 rich      rich      0 Oct 17 21:30 testing.UfIi13
-rw----- 1 rich      rich      0 Oct 17 21:57 testing.lVBtkW
$
```

As you can see, the output of the `mktemp` command is the name of the file that it creates. When you use the `mktemp` command in a script, you'll want to save that filename in a variable, so you can refer to it later on in the script:

```
$ cat test19
#!/bin/bash
# creating and using a temp file

tempfile=$(mktemp test19.XXXXXX)

exec 3>$tempfile

echo "This script writes to temp file $tempfile"

echo "This is the first line" >&3
echo "This is the second line." >&3
echo "This is the last line." >&3
exec 3>&-

echo "Done creating temp file. The contents are:"
cat $tempfile
rm -f $tempfile 2> /dev/null
$ ./test19
This script writes to temp file test19.vCHoya
Done creating temp file. The contents are:
This is the first line
This is the second line.
This is the last line.
$ ls -al test19*
-rwxr--r-- 1 rich      rich      356 Oct 29 22:03 test19*
$
```

The script uses the `mktemp` command to create a temporary file and assigns the filename to the `$tempfile` variable. It then uses the temporary file as the output redirection file for file descriptor 3. After displaying the temporary filename on `STDOUT`, it writes a few lines to the temporary file, and then it closes the file descriptor. Finally, it displays the contents of the temporary file and then uses the `rm` command to remove it.

Creating a temporary file in /tmp

The `-t` option forces `mktemp` to create the file in the temporary directory of the system. When you use this feature, the `mktemp` command returns the full pathname used to create the temporary file, not just the filename:

```
$ mktemp -t test.XXXXXX
/tmp/test.xG3374
$ ls -al /tmp/test*
-rw----- 1 rich rich 0 2014-10-29 18:41 /tmp/test.xG3374
$
```

Because the `mktemp` command returns the full pathname, you can then reference the temporary file from any directory on the Linux system, no matter where it places the temporary directory:

```
$ cat test20
#!/bin/bash
# creating a temp file in /tmp

tempfile=$(mktemp -t tmp.XXXXXX)

echo "This is a test file." > $tempfile
echo "This is the second line of the test." >> $tempfile

echo "The temp file is located at: $tempfile"
cat $tempfile
rm -f $tempfile
$ ./test20
The temp file is located at: /tmp/tmp.Ma3390
This is a test file.
This is the second line of the test.
$
```

When `mktemp` creates the temporary file, it returns the full pathname to the environment variable. You can then use that value in any command to reference the temporary file.

Creating a temporary directory

The `-d` option tells the `mktemp` command to create a temporary directory instead of a file. You can then use that directory for whatever purposes you need, such as creating additional temporary files:

```
$ cat test21
#!/bin/bash
# using a temporary directory

tempdir=$(mktemp -d dir.XXXXXX)
cd $tempdir
tempfile1=$(mktemp temp.XXXXXX)
tempfile2=$(mktemp temp.XXXXXX)
exec 7> $tempfile1
exec 8> $tempfile2

echo "Sending data to directory $tempdir"
echo "This is a test line of data for $tempfile1" >&7
echo "This is a test line of data for $tempfile2" >&8
$ ./test21
Sending data to directory dir.ouT8S8
$ ls -al
total 72
drwxr-xr-x   3 rich   rich   4096 Oct 17 22:20 ./
drwxr-xr-x   9 rich   rich   4096 Oct 17 09:44 ../
drwx-----  2 rich   rich   4096 Oct 17 22:20 dir.ouT8S8/
-rwxr--r--   1 rich   rich    338 Oct 17 22:20 test21*
$ cd dir.ouT8S8
[dir.ouT8S8]$ ls -al
total 16
drwx-----  2 rich   rich   4096 Oct 17 22:20 ./
drwxr-xr-x   3 rich   rich   4096 Oct 17 22:20 ../
-rw-----   1 rich   rich    44 Oct 17 22:20 temp.N5F3O6
-rw-----   1 rich   rich    44 Oct 17 22:20 temp.SQslb7
[dir.ouT8S8]$ cat temp.N5F3O6
This is a test line of data for temp.N5F3O6
[dir.ouT8S8]$ cat temp.SQslb7
This is a test line of data for temp.SQslb7
[dir.ouT8S8]$
```

The script creates a directory in the current directory and uses the `cd` command to change to that directory before creating two temporary files. The two temporary files are then assigned to file descriptors and used to store output from the script.

Logging Messages

Sometimes, it's beneficial to send output both to the monitor and to a file for logging. Instead of having to redirect output twice, you can use the special `tee` command.

The `tee` command is like a T-connector for pipes. It sends data from `STDIN` to two destinations at the same time. One destination is `STDOUT`. The other destination is a filename specified on the `tee` command line:

```
tee filename
```

Because `tee` redirects data from `STDIN`, you can use it with the pipe command to redirect output from any command:

```
$ date | tee testfile
Sun Oct 19 18:56:21 EDT 2014
$ cat testfile
Sun Oct 19 18:56:21 EDT 2014
$
```

The output appears in `STDOUT` and is written to the file specified. Be careful: By default, the `tee` command overwrites the output file on each use:

```
$ who | tee testfile
rich      pts/0      2014-10-17 18:41 (192.168.1.2)
$ cat testfile
rich      pts/0      2014-10-17 18:41 (192.168.1.2)
$
```

If you want to append data to the file, you must use the `-a` option:

```
$ date | tee -a testfile
Sun Oct 19 18:58:05 EDT 2014
$ cat testfile
rich      pts/0      2014-10-17 18:41 (192.168.1.2)
Sun Oct 19 18:58:05 EDT 2014
$
```

Using this technique, you can both save data in files and display the data on the monitor for your users:

```
$ cat test22
#!/bin/bash
# using the tee command for logging

tempfile=test22file

echo "This is the start of the test" | tee $tempfile
echo "This is the second line of the test" | tee -a $tempfile
echo "This is the end of the test" | tee -a $tempfile
$ ./test22
This is the start of the test
This is the second line of the test
This is the end of the test
$ cat test22file
This is the start of the test
This is the second line of the test
This is the end of the test
$
```

Now you can save a permanent copy of your output at the same time as you're displaying it to your users.

Practical Example

File redirection is very common both when reading files into scripts and when outputting data from a script into a file. This example script does both of those things. It reads a .csv-formatted data file and outputs SQL `INSERT` statements to insert the data into a database (see Chapter 25).

The shell script uses a command line parameter to define the name of the .csv file from which to read the data. The .csv format is used to export data from spreadsheets, so you can place the database data into a spreadsheet, save the spreadsheet in .csv format, read the file, and create `INSERT` statements to insert the data into a MySQL database.

Here's what the script looks like:

```
$cat test23
#!/bin/bash
# read file and create INSERT statements for MySQL

outfile='members.sql'
IFS=','
while read lname fname address city state zip
do
    cat >> $outfile << EOF
        INSERT INTO members (lname,fname,address,city,state,zip) VALUES
        ('$lname', '$fname', '$address', '$city', '$state', '$zip');
    EOF
done < ${1}
$
```

That's a pretty short script, thanks to the file redirection that goes on! There are three redirection operations happening in the script. The `while` loop uses the `read` statement (discussed in Chapter 14) to read text from the data file. Notice in the `done` statement the redirection symbol:

```
done < ${1}
```

The `$1` represents the first command line parameter when you run the `test23` program. That specifies the data file from which to read the data. The `read` statement parses the text using the `IFS` character, which we specify as a comma.

The other two redirection operations in the script both appear in the same statement:

```
cat >> $outfile << EOF
```

This one statement has one output append redirection (the double greater-than symbol) and one input append redirection (the double less-than symbol). The output redirection appends the `cat` command output to the file specified by the `$outfile` variable. The input to the `cat` command is redirected from the standard input to use the data stored inside the script. The EOF symbol marks the start and end delimiter of the data that's appended to the file:

```
INSERT INTO members (lname,fname,address,city,state,zip) VALUES
('$lname', '$fname', '$address', '$city', '$state', '$zip');
```

The text creates a standard SQL `INSERT` statement. Notice that the data values are replaced with the variables for the data read from the `read` statement.

So basically the `while` loop reads on the data one line at a time, plugs those data values into the `INSERT` statement template, then outputs the result to the output file.

For this experiment, I used this as the input data file:

```
$ cat members.csv
Blum,Richard,123 Main St.,Chicago,IL,60601
Blum,Barbara,123 Main St.,Chicago,IL,60601
Bresnahan,Christine,456 Oak Ave.,Columbus,OH,43201
Bresnahan,Timothy,456 Oak Ave.,Columbus,OH,43201
$
```

When you run the script, nothing appears in the output on the monitor:

```
$ ./test23 < members.csv
$
```

But when you look at the `members.sql` output file, you should see the output data:

```
$ cat members.sql
INSERT INTO members (lname,fname,address,city,state,zip) VALUES ('Blum',
'Richard', '123 Main St.', 'Chicago', 'IL', '60601');
INSERT INTO members (lname,fname,address,city,state,zip) VALUES ('Blum',
'Barbara', '123 Main St.', 'Chicago', 'IL', '60601');
INSERT INTO members (lname,fname,address,city,state,zip) VALUES ('Bresnahan',
'Christine', '456 Oak Ave.', 'Columbus', 'OH', '43201');
INSERT INTO members (lname,fname,address,city,state,zip) VALUES ('Bresnahan',
'Timothy', '456 Oak Ave.', 'Columbus', 'OH', '43201');
$
```

The script worked exactly as expected! Now you can easily import the `members.sql` file into a MySQL database table (see Chapter 25).

Summary

Understanding how the bash shell handles input and output can come in handy when creating your scripts. You can manipulate both how the script receives data and how it displays data, to customize your script for any environment. You can redirect the input of a script from the standard input (STDIN) to any file on the system. You can also redirect the output of the script from the standard output (STDOUT) to any file on the system.

Besides the `STDOUT`, you can redirect any error messages your script generates by redirecting the `STDERR` output. This is accomplished by redirecting the file descriptor associated with the `STDERR` output, which is file descriptor 2. You can redirect `STDERR` output to the same file as the `STDOUT` output or to a completely separate file. This enables you to separate normal script messages from any error messages generated by the script.

The bash shell allows you to create your own file descriptors for use in your scripts. You can create file descriptors 3 through 8 and assign them to any output file you desire. After you create a file descriptor, you can redirect the output of any command to it, using the standard redirection symbols.

The bash shell also allows you to redirect input to a file descriptor, providing an easy way to read data contained in a file into your script. You can use the `ls -l` command to display the active file descriptors in your shell.

Linux systems provide a special file, called `/dev/null`, to allow you to redirect output that you don't want. The Linux system discards anything redirected to the `/dev/null` file. You can also use this file to produce an empty file by redirecting the contents of the `/dev/null` file to the file.

The `mktemp` command is a handy feature of the bash shell that allows you to easily create temporary files and directories. Simply specify a template for the `mktemp` command, and it creates a unique file each time you call it, based on the file template format. You can also create temporary files and directories in the `/tmp` directory on the Linux system, which is a special location that isn't preserved between system boots.

The `tee` command is a handy way to send output both to the standard output and to a log file. This enables you to display messages from your script on the monitor and store them in a log file at the same time.

In Chapter 16, you'll see how to control and run your scripts. Linux provides several different methods for running scripts other than directly from the command line interface prompt. You'll see how to schedule your scripts to run at a specific time, as well as learn how to pause them while they're running.