In this chapter:
- *Standard Input and Standard Output*
- *Pipes and Filters*

5

Redirecting I/O

Many Unix programs read input (such as a file) and write output. In this chapter, we discuss Unix programs that handle their input and output in a standard way. This lets them work with each other.

This chapter generally *doesn't* apply to full-screen programs, such as the Pico editor, that take control of your whole terminal window. (The pager programs, **less**, **more**, and **pg**, *do* work together in this way.) It also doesn't apply to graphical programs, such as StarOffice or Netscape, that open their own windows on your screen.

Standard Input and Standard Output

What happens if you don't give a filename argument on a command line? Most programs will take their input from your keyboard instead (after you press the first RETURN to start the program running, that is). Your terminal keyboard is the program's *standard input.*

As a program runs, the results are usually displayed on your terminal screen. The terminal screen is the program's *standard output.*

So, by default, each of these programs takes its input from the standard input and sends the results to the standard output.

These two default cases of input/output (I/O) can be varied. This is called *I/O redirection.*

If a program doesn't normally read from files, but reads from its standard input, you can give a filename by using the < (less-than symbol) operator.

For example, the **mail** program (see the section "Sending Mail from a Shell Prompt" in Chapter 6) normally reads the message to send from your keyboard. Here's how to use the input redirection operator to mail the contents of the file *to_do* to *bigboss@corp.xyz*:

```
$ mail bigboss@corp.xyz < to_do
$
```

If a program writes to its standard output, which is normally the screen, you can make it write to a file instead by using the greater-than symbol (>) operator. The pipe operator (|) sends the standard output of one program to the standard input of another program. Input/output redirection is one of the most powerful and flexible Unix features, We'll take a closer look at it soon.

Putting Text in a File

Instead of always letting a program's output come to the screen, you can redirect output into a file. This is useful when you'd like to save program output or when you put files together to make a bigger file.

cat

cat, which is short for "concatenate," reads files and outputs their contents one after another, without stopping.

To display files on the standard output (your screen), use:

cat *file(s)*

For example, let's display the contents of the file */etc/passwd*. This system file describes users' accounts. (Your system may have a more complete list somewhere else.)

```
$ cat /etc/passwd
root:x&k8KP30f;(:0:0:Root:/:
daemon:*:1:1:Admin:/:
        .
        .
        .
john::128:50:John Doe:/usr/john:/bin/sh
$
```

You cannot go back to view the previous screens, as you can when you use a pager program such as **less** (unless you're using a terminal window with a scrollbar, that is). **cat** is mainly used with redirection, as we'll see in a moment.

By the way: if you enter **cat** without a filename, it tries to read from the keyboard (as we mention earlier). You can get out by pressing RETURN followed by a single CTRL-D .

The > operator

When you add "> *filename*" to the end of a command line, the program's output is diverted from the standard output to the named file. The > symbol is called the *output redirection operator*.

When you use the > operator, be careful not to accidentally overwrite a file's contents. Your system may let you redirect output to an existing file. If so, the old file will be deleted (or, in Unix lingo, "clobbered"). Be careful not to overwrite a much needed file!

Many shells can protect you from this risk. In the C shell, use the command **set noclobber**. The Korn shell and **bash** command is **set -o noclobber**. Enter the command at a shell prompt or put it in your shell's startup file. After that, the shell does not allow you to redirect onto an existing file and overwrite its contents.

This doesn't protect against overwriting by Unix programs such as **cp**; it works only with the > redirection operator. For more protection, you can set Unix file access permissions.

For example, let's use **cat** with this operator. The file contents that you'd normally see on the screen (from the standard output) are diverted into another file, which we'll then read using **cat** (without any redirection!):

```
$ cat /etc/passwd > password
$ cat password
root:x&k8KP30f;(:0:0:Root:/:
daemon:*:1:1:Admin:/:
        .
        .
        .
john::128:50:John Doe:/usr/john:/bin/sh
$
```

An earlier example (in the section "cat") showed how **cat /etc/passwd** displays the file */etc/passwd* on the screen. The example here adds the >

operator; so the output of **cat** goes to a file called *password* in the working directory. Displaying the file *password* shows that its contents are the same as the file */etc/passwd* (the effect is the same as the copy command **cp /etc/passwd password**).

You can use the > redirection operator with any program that sends text to its standard output—not just with **cat**. For example:

```
$ who > users
$ date > today
$ ls
password   today   users   ...
```

We've sent the output of **who** to a file called *users* and the output of **date** to the file named *today*. Listing the directory shows the two new files. Let's look at the output from the **who** and **date** programs by reading these two files with **cat**:

```
$ cat users
tim     tty1    Aug 12  07:30
john    tty4    Aug 12  08:26
$ cat today
Tue Aug 12 08:36:09 EDT 2001
$
```

You can also use the **cat** program and the > operator to make a small text file. We told you earlier to type CTRL-D if you accidentally enter **cat** without a filename. This is because the **cat** program alone takes whatever you type on the keyboard as input. Thus, the command:

cat > *filename*

takes input from the keyboard and redirects it to a file. Try the following example:

```
$ cat > to_do
Finish report by noon
Lunch with Xannie
Swim at 5:30
^D
$
```

cat takes the text that you typed as input (in this example, the three lines that begin with **Finish**, **Lunch**, and **Swim**), and the > operator redirects it to a file called *to_do*. Type CTRL-D *once*, on a new line by itself, to signal the end of the text. You should get a shell prompt.

You can also create a bigger file from smaller files with the **cat** command and the > operator. The form:

cat *file1 file2* > *newfile*

creates a file *newfile*, consisting of *file1* followed by *file2*.

```
$ cat today to_do > diary
$ cat diary
Tue Aug 12 08:36:09 EDT 2001
Finish report by noon
Lunch with Xannie
Swim at 5:30
$
```

You can't use redirection to add a file to itself, along with other files. For example, you might hope that the following command would merge today's to-do list with tomorrow's. This won't work!

```
$ cat to_do to_do.tomorrow > to_do.tomorrow
cat: to_do.tomorrow: input file is output file
```

cat warns you, but it's actually already too late. When you redirect a program's output to a file, Unix empties (clobbers) the file *before* the program starts running. The right way to do this is by using a temporary file (as you'll see in a later example) or simply by using a text editor program.

The >> operator

You can add more text to the end of an existing file, instead of replacing its contents, by using the >> (append redirection) operator. Use it as you would the > (output redirection) operator. So:

cat *file2* >> *file1*

appends the contents of *file2* to the end of *file1*. For an example, let's append the contents of the file *users*, and also the current date and time, to the file *diary*. Then we display the file:

```
$ cat users >> diary
$ date >> diary
$ cat diary
Tue Aug 12 08:36:09 EDT 2001
```

```
Finish report by noon
Lunch with Xannie
Swim at 5:30
tim     tty1    Aug 12  07:30
john    tty4    Aug 12  08:26
Tue Aug 12 09:07:24 EDT 2001
$
```

Unix doesn't have a redirection operator that adds text to the beginning of a file. You can do this by storing the new text in a temporary file, then by using a text editor program to read the temporary file into the start of the file you want to edit. You also can do the job with a temporary file and redirection. Maybe you'd like each day's entry to go at the beginning of your *diary* file. Simply rename *diary* to something like *temp*. Make a new *diary* file with today's entries, then append *temp* (with its old contents) to the new *diary*. For example:*

```
$ mv diary temp
$ date > diary
$ cat users >> diary
$ cat temp >> diary
$ rm temp
```

Pipes and Filters

We've seen how to redirect input from a file and output to a file. You can also connect two *programs* together so that the output from one program becomes the input of the next program. Two or more programs connected in this way form a *pipe*. To make a pipe, put a vertical bar (|) on the command line between two commands. When a pipe is set up between two commands, the standard output of the command to the left of the pipe symbol becomes the standard input of the command to the right of the pipe symbol. Any two commands can form a pipe as long as the first program writes to standard output and the second program reads from standard input.

When a program takes its input from another program, performs some operation on that input, and writes the result to the standard output (which may be piped to yet another program), it is referred to as a *filter*. A common use of filters is to modify output. Just as a common filter culls unwanted items, Unix filters can restructure output.

* This example could be shortened by combining the two **cat** commands into one, giving both filenames as arguments to a single **cat** command. That wouldn't work, though, if you were making a real diary with a command other than **cat users**.

Most Unix programs can be used to form pipes. Some programs that are commonly used as filters are described in the next sections. Note that these programs aren't used only as filters or parts of pipes. They're also useful on their own.

grep

The **grep** program searches a file or files for lines that have a certain pattern. The syntax is:

grep *"pattern" file(s)*

The name "grep" derives from the **ed** (a Unix line editor) command **g/re/p**, which means "*g*lobally search for a *r*egular *e*xpression and *p*rint all lines containing it." A *regular expression* is either some plain text (a word, for example) and/or special characters used for pattern matching. When you learn more about regular expressions, you can use them to specify complex patterns of text.

The simplest use of **grep** is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. But let's start with an example reading from files: searching all files in the working directory for a word—say, *Unix*. We'll use the wildcard * to quickly give **grep** all filenames in the directory.

```
$ grep "Unix" *
ch01:Unix is a flexible and powerful operating system
ch01:When the Unix designers started work, little did
ch05:What can we do with Unix?
$
```

When **grep** searches multiple files, it shows the filename where it finds each matching line of text. Alternatively, if you don't give **grep** a filename to read, it reads its standard input; that's the way all filter programs work:

```
$ ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

First, the example runs **ls -l** to list your directory. The standard output of **ls -l** is piped to **grep**, which only outputs lines that contain the string *Aug* (that is, files that were last modified in August). Because the standard output of **grep** isn't redirected, those lines go to the terminal screen.

grep options let you modify the search. Table 1-1 lists some of the options.

Table 5-1. Some grep options

Option	Description
-v	Print all lines that do not match pattern.
-n	Print the matched line and its line number.
-l	Print only the names of files with matching lines (lowercase letter "L").
-c	Print only the count of matching lines.
-i	Match either upper- or lowercase.

Next, let's use a regular expression that tells **grep** to find lines with *carol*, followed by zero or more other characters (abbreviated in a regular expression as "`.*`"),* then followed by *Aug*:

```
$ ls -l | grep "carol.*Aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

For more about regular expressions, see the references in the section "Documentation" (Chapter 8).

sort

The **sort** program arranges lines of text alphabetically or numerically. The following example sorts the lines in the *food* file (from the section "Printing Files" in Chapter 4) alphabetically. **sort** doesn't modify the file itself; it reads the file and writes the sorted text to the standard output.

```
$ sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
```

* Note that the regular expression for "zero or more characters," "`.*`", is different than the corresponding filename wildcard "`*`". See the section "File and Directory Wildcards" in Chapter 4. We can't cover regular expressions in enough depth here to explain the difference—though more-detailed books do. As a rule of thumb, remember that the first argument to **grep** is a regular expression; other arguments, if any, are filenames that can use wildcards.

By default, **sort** arranges lines of text alphabetically. Many options control the sorting, and Table 1-2 lists some of them.

Table 5-2. Some sort options

Option	Description
-n	Sort numerically (example: 10 sorts after 2), ignore blanks and tabs.
-r	Reverse the sorting order.
-f	Sort upper- and lowercase together.
+*x*	Ignore first *x* fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by order of size. The following pipe uses the commands **ls**, **grep**, and **sort**:

```
$ ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by order of size, and prints them to the terminal screen. The **sort** option **+4n** skips four fields (fields are separated by blanks), then sorts the lines in numeric order. So, the output of **ls**, filtered by **grep**, is sorted by the file size (this is the fifth column, starting with 1605). Both **grep** and **sort** are used here as filters to modify the output of the **ls -l** command. If you wanted to email this listing to someone, you could add a final pipe to the **mail** program. Or you could print the listing by piping the **sort** output to your printer command (either **lp** or **lpr**).

Piping to a Pager

The **less** program, which you saw in the section "Looking Inside Files with less" in Chapter 3, can also be used as a filter. A long output normally zips by you on the screen, but if you run text through **less**, the display stops after each screenful of text.

Let's assume that you have a long directory listing. (If you want to try this example and need a directory with lots of files, use **cd** first to change to a

system directory such as */bin* or */usr/bin.*) To make it easier to read the sorted listing, pipe the output through **less**:

```
$ ls -l | grep "Aug" | sort +4n | less
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc     14827 Aug  9 12:40 ch03
        .
        .
        .
-rw-rw-rw-  1 john  doc     16867 Aug  6 15:56 ch05
:
```

less reads a screenful of text from the pipe (consisting of lines sorted by order of file size), then prints a colon (**:**) prompt. At the prompt, you can type a **less** command to move through the sorted text. **less** reads more text from the pipe and shows it to you, as well as saves a copy of what it has read, so you can go backwards to reread previous text if you want to. (The simpler pager programs **more** and **pg** generally can't back up while reading from a pipe.) When you're done seeing the sorted text, the **q** command quits **less**.

Exercise: redirecting input/output

In the following exercises you redirect output, create a simple pipe, and use filters to modify output.

Redirect output to a file.	Enter **who > users**
Email that file to yourself. (Replace *username* with your own username.)	Enter **mail** *username* **< users**
Sort output of a program.	Enter **who | sort**
Append sorted output to a file.	Enter **who | sort >> users**
Display output to screen.	Enter **less users** (or **more users** or **pg users**)
Display long output to screen.	Enter **ls -l /bin | less** (or **more** or **pg**)
Format and print a file with **pr**.	Enter **pr users | lp** or **pr users | lpr**