20

TEXT PROCESSING

All Unix-like operating systems rely heavily on text files for data storage. So it makes sense that there are many tools for manipulating text. In this chapter, we will look at programs that are used to "slice and dice" text. In the next chapter, we will look at more text processing, focusing on programs that are used to format text for printing and other kinds of human consumption.

This chapter will revisit some old friends and introduce us to some new ones:

`cat` Concatenate files and print on the standard output

`sort` Sort lines of text files

`uniq` Report or omit repeated lines

`cut` Remove sections from each line of files

`paste` Merge lines of files

`join` Join lines of two files on a common field

`comm` Compare two sorted files line by line

`diff` Compare files line by line

`patch` Apply a diff file to an original

`tr` Translate or delete characters

`sed` Stream editor for filtering and transforming text

`aspell` Interactive spellchecker

Applications of Text

So far, we have learned a couple of text editors (`nano` and `vim`), looked at a bunch of configuration files, and have witnessed the output of dozens of commands, all in text. But what else is text used for? For many things, it turns out.

Documents

Many people write documents using plain text formats. While it is easy to see how a small text file could be useful for keeping simple notes, it is also possible to write large documents in text format. One popular approach is to write a large document in a text format and then embed a *markup language* to describe the formatting of the finished document. Many scientific papers are written using this method, as Unix-based text processing systems were among the first systems that supported the advanced typographical layout needed by writers in technical disciplines.

Web Pages

The world's most popular type of electronic document is probably the web page. Web pages are text documents that use either *Hypertext Markup Language* (*HTML*) or *Extensible Markup Language* (*XML*) as markup languages to describe the document's visual format.

Email

Email is an intrinsically text-based medium. Even non-text attachments are converted into a text representation for transmission. We can see this for ourselves by downloading an email message and then viewing it in `less`. We will see that the message begins with a *header* that describes the source of the message and the processing it received during its journey, followed by the *body* of the message with its content.

Printer Output

On Unix-like systems, output destined for a printer is sent as plain text or, if the page contains graphics, is converted into a text format *page description language* known as *PostScript*, which is then sent to a program that generates the graphic dots to be printed.

Program Source Code

Many of the command line programs found on Unix-like systems were created to support system administration and software development, and text processing programs are no exception. Many of them are designed to solve software development problems. The reason text processing is important to software developers is that all software starts out as text. *Source code*, the part of the program the programmer actually writes, is always in text format.

Revisiting Some Old Friends

Back in Chapter 6 we learned about some commands that are able to accept standard input in addition to command line arguments. We touched on them only briefly then, but now we will take a closer look at how they can be used to perform text processing.

cat

The `cat` program has a number of interesting options. Many of them are used to help better visualize text content. One example is the `-A` option, which is used to display non-printing characters in the text. There are times when we want to know whether control characters are embedded

in our otherwise visible text. The most common of these are tab characters (as opposed to spaces) and carriage returns, often present as end-of-line characters in MS-DOS-style text files. Another common situation is a file containing lines of text with trailing spaces.

Let's create a test file using `cat` as a primitive word processor. To do this, we'll just enter the command `cat` (along with specifying a file for redirected output) and type our text, followed by ENTER to properly end the line and then CTRL-D to indicate to `cat` that we have reached end of file. In this example, we enter a leading tab character and follow the line with some trailing spaces:

```
[me@linuxbox ~]$ cat > foo.txt
   The quick brown fox jumped over the lazy dog.
[me@linuxbox ~]$
```

Next, we use `cat` with the `-A` option to display the text.

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog.   $
[me@linuxbox ~]$
```

As we can see in the results, the tab character in our text is represented by `^I`. This is a common notation that means CTRL-I, which, as it turns out, is the same as a tab character. We also see that a `$` appears at the true end of the line, indicating that our text contains trailing spaces.

MS-DOS TEXT VS. UNIX TEXT

One of the reasons you may want to use `cat` to look for non-printing characters in text is to spot hidden carriage returns. Where do hidden carriage returns come from? DOS and Windows! Unix and DOS don't define the end of a line the same way in text files. Unix ends a line with a linefeed character (ASCII 10), while MS-DOS and its derivatives use the sequence carriage return (ASCII 13) and linefeed to terminate each line of text.

There are several ways to convert files from DOS to Unix format. On many Linux systems, there are programs called `dos2unix` and `unix2dos`, which can convert text files to and from DOS format. However, if you don't have `dos2unix` on your system, don't worry. The process of converting text from DOS to Unix format is simple; it involves the removal of the offending carriage returns. That is easily accomplished by a couple of the programs discussed later in this chapter.

`cat` also has options that are used to modify text. The two most prominent are `-n`, which numbers lines, and `-s`, which suppresses the output of multiple blank lines. We can demonstrate thusly:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox


jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
   1  The quick brown fox
   2
   3  jumped over the lazy dog.
[me@linuxbox ~]$
```

In this example, we create a new version of our *foo.txt* test file, which contains two lines of text separated by two blank lines. After processing by `cat` with the `-ns` options, the extra blank line is removed, and the remaining lines are numbered. While this is not much of a process to perform on text, it is a process.

sort

The `sort` program sorts the contents of standard input, or one or more files specified on the command line, and sends the results to standard output. Using the same technique that we used with `cat`, we can demonstrate processing of standard input directly from the keyboard as follows:

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

After entering the command, we enter the letters *c*, *b*, and *a*, and then we press CTRL-D to indicate end of file. We then view the resulting file and see that the lines now appear in sorted order.

Because `sort` can accept multiple files on the command line as arguments, it is possible to *merge* multiple files into a single sorted whole. For example, if we had three text files and wanted to combine them into a single sorted file, we could do something like this:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

`sort` has several interesting options. Table 20-1 provides a partial list.

Table 20-1: Common `sort` Options

Option	Long option	Description
`-b`	`--ignore-leading-blanks`	By default, sorting is performed on the entire line, starting with the first character in the line. This option causes `sort` to ignore leading spaces in lines and calculates sorting based on the first non-whitespace character on the line.
`-f`	`--ignore-case`	Make sorting case-insensitive.

Option	Long option	Description
`-n`	`--numeric-sort`	Perform sorting based on the numeric evaluation of a string. Using this option allows sorting to be performed on numeric values rather than alphabetic values.
`-r`	`--reverse`	Sort in reverse order. Results are in descending rather than ascending order.
`-k`	`--key=field1[,field2]`	Sort based on a key field located from `field1` to `field2` rather than the entire line. See the following discussion.
`-m`	`--merge`	Treat each argument as the name of a presorted file. Merge multiple files into a single sorted result without performing any additional sorting.
`-o`	`--output=file`	Send sorted output to `file` rather than standard output.
`-t`	`--field-separator=char`	Define the field-separator character. By default fields are separated by spaces or tabs.

Although most of these options are pretty self-explanatory, some are not. First, let's look at the `-n` option, used for numeric sorting. With this option, it is possible to sort values based on numeric values rather than lexographically. We can demonstrate this by sorting the results of the `du` command to determine the largest users of disk space. Normally, the `du` command lists the results of a summary in pathname order.

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252     /usr/share/aclocal
96      /usr/share/acpi-support
8       /usr/share/adduser
196     /usr/share/alacarte
344     /usr/share/alsa
8       /usr/share/alsa-base
12488   /usr/share/anthy
8       /usr/share/apmdq
21440   /usr/share/app-install
48      /usr/share/application-registry
```

In this example, we pipe the results into `head` to limit the results to the first 10 lines. We can produce a numerically sorted list to show the 10 largest consumers of space this way.

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940  /usr/share/locale-langpack
242660  /usr/share/doc
197560  /usr/share/fonts
179144  /usr/share/gnome
146764  /usr/share/myspell
144304  /usr/share/gimp
135880  /usr/share/dict
76508   /usr/share/icons
68072   /usr/share/apps
62844   /usr/share/foomatic
```

By using the `n` and `r` options, we produce a reverse numerical sort, with the largest values appearing first in the results. This sort works because the numerical values occur at the beginning of each line. But what if we want to sort a list based on some value found within the line? For example, here are the results of `ls -l`:

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
```

```
-rwxr-xr-x 1 root   root      34824 2016-04-04 02:42 [
-rwxr-xr-x 1 root   root     101556 2007-11-27 06:08 a2p
-rwxr-xr-x 1 root   root      13036 2016-02-27 08:22 aconnect
-rwxr-xr-x 1 root   root      10552 2007-08-15 10:34 acpi
-rwxr-xr-x 1 root   root       3800 2016-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root   root       7536 2016-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root   root       3576 2016-04-29 07:57 addpart
-rwxr-xr-x 1 root   root      20808 2016-01-03 18:02 addr2line
-rwxr-xr-x 1 root   root     489704 2016-10-09 17:02 adept_batch
```

Ignoring, for the moment, that `ls` can sort its results by size, we could use `sort` to sort this list by file size, as well.

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nrk 5 | head
-rwxr-xr-x 1 root   root    8234216 2016-04-07 17:42 inkscape
-rwxr-xr-x 1 root   root    8222692 2016-04-07 17:42 inkview
-rwxr-xr-x 1 root   root    3746508 2016-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root   root    3654020 2016-08-26 16:16 quanta
-rwxr-xr-x 1 root   root    2928760 2016-09-10 14:31 gdbtui
-rwxr-xr-x 1 root   root    2928756 2016-09-10 14:31 gdb
-rwxr-xr-x 1 root   root    2602236 2016-10-10 12:56 net
-rwxr-xr-x 1 root   root    2304684 2016-10-10 12:56 rpcclient
-rwxr-xr-x 1 root   root    2241832 2016-04-04 05:56 aptitude
-rwxr-xr-x 1 root   root    2202476 2016-10-10 12:56 smbcacls
```

Many uses of `sort` involve the processing of *tabular data*, such as the results of the previous `ls` command. If we apply database terminology to the previous table, we would say that each row is a *record* and that each record consists of multiple *fields*, such as the file attributes, link count, filename, file size, and so on. `sort` is able to process individual fields. In database terms, we are able to specify one or more *key fields* to use as *sort keys*. In the previous example, we specify the `n` and `r` options to perform a reverse numerical sort and specify `-k 5` to make `sort` use the fifth field as the key for sorting.

The `k` option is interesting and has many features, but first we need to talk about how `sort` defines fields. Let's consider the following simple text file consisting of a single line containing the author's name:

```
William Shotts
```

By default, `sort` sees this line as having two fields. The first field contains these characters: `"William"`. The second field contains these characters: `"Shotts"`.

This means that whitespace characters (spaces and tabs) are used as delimiters between fields and that the delimiters are included in the field when sorting is performed.

Looking again at a line from our `ls` output, as follows, we can see that a line contains eight fields and that the fifth field is the file size:

```
-rwxr-xr-x 1 root   root    8234216 2016-04-07 17:42 inkscape
```

For our next series of experiments, let's consider the following file containing the history of three popular Linux distributions released from 2006 to 2008. Each line in the file has three fields: the distribution name, version number, and date of release in MM/DD/YYYY format.

```
SUSE      10.2       12/07/2006
Fedora    10         11/25/2008
SUSE      11.0       06/19/2008
Ubuntu    8.04       04/24/2008
Fedora    8          11/08/2007
SUSE      10.3       10/04/2007
Ubuntu    6.10       10/26/2006
Fedora    7          05/31/2007
Ubuntu    7.10       10/18/2007
Ubuntu    7.04       04/19/2007
SUSE      10.1       05/11/2006
Fedora    6          10/24/2006
Fedora    9          05/13/2008
```

```
Ubuntu    6.06       06/01/2006
Ubuntu    8.10       10/30/2008
Fedora    5          03/20/2006
```

Using a text editor (perhaps `vim`), we'll enter this data and name the resulting file *distros.txt*.

Next, we'll try sorting the file and observe these results:

```
[me@linuxbox ~]$ sort distros.txt
Fedora    10         11/25/2008
Fedora    5          03/20/2006
Fedora    6          10/24/2006
Fedora    7          05/31/2007
Fedora    8          11/08/2007
Fedora    9          05/13/2008
SUSE      10.1       05/11/2006
SUSE      10.2       12/07/2006
SUSE      10.3       10/04/2007
SUSE      11.0       06/19/2008
Ubuntu    6.06       06/01/2006
Ubuntu    6.10       10/26/2006
Ubuntu    7.04       04/19/2007
Ubuntu    7.10       10/18/2007
Ubuntu    8.04       04/24/2008
Ubuntu    8.10       10/30/2008
```

Well, it mostly worked. The problem occurs in the sorting of the Fedora version numbers. Because 1 comes before 5 in the character set, version 10 ends up at the top while version 9 falls to the bottom.

To fix this problem, we are going to have to sort on multiple keys. We want to perform an alphabetic sort on the first field and then a numeric sort on the second field. `sort` allows multiple instances of the `-k` option so that multiple sort keys can be specified. In fact, a key may include a range of fields. If no range is specified (as has been the case with our previous

examples), `sort` uses a key that begins with the specified field and extends to the end of the line. Here is the syntax for our multikey sort:

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora    5        03/20/2006
Fedora    6        10/24/2006
Fedora    7        05/31/2007
Fedora    8        11/08/2007
Fedora    9        05/13/2008
Fedora    10       11/25/2008
SUSE      10.1     05/11/2006
SUSE      10.2     12/07/2006
SUSE      10.3     10/04/2007
SUSE      11.0     06/19/2008
Ubuntu    6.06     06/01/2006
Ubuntu    6.10     10/26/2006
Ubuntu    7.04     04/19/2007
Ubuntu    7.10     10/18/2007
Ubuntu    8.04     04/24/2008
Ubuntu    8.10     10/30/2008
```

Though we used the long form of the option for clarity, `-k 1,1 -k 2n` would be exactly equivalent. In the first instance of the key option, we specified a range of fields to include in the first key. Because we wanted to limit the sort to just the first field, we specified `1,1`, which means "start at field 1 and end at field 1." In the second instance, we specified `2n`, which means field 2 is the sort key and that the sort should be numeric. An option letter may be included at the end of a key specifier to indicate the type of sort to be performed. These option letters are the same as the global options for the `sort` program: `b` (ignore leading blanks), `n` (numeric sort), `r` (reverse sort), and so on.

The third field in our list contains a date in an inconvenient format for sorting. On computers, dates are usually formatted in YYYY-MM-DD order to make chronological sorting easy, but ours are in the American format of MM/DD/YYYY. How can we sort this list in chronological order?

Fortunately, `sort` provides a way. The key option allows specification of *offsets* within fields, so we can define keys within fields.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora    10        11/25/2008
Ubuntu    8.10      10/30/2008
SUSE      11.0      06/19/2008
Fedora    9         05/13/2008
Ubuntu    8.04      04/24/2008
Fedora    8         11/08/2007
Ubuntu    7.10      10/18/2007
SUSE      10.3      10/04/2007
Fedora    7         05/31/2007
Ubuntu    7.04      04/19/2007
SUSE      10.2      12/07/2006
Ubuntu    6.10      10/26/2006
Fedora    6         10/24/2006
Ubuntu    6.06      06/01/2006
SUSE      10.1      05/11/2006
Fedora    5         03/20/2006
```

By specifying `-k 3.7`, we instruct `sort` to use a sort key that begins at the seventh character within the third field, which corresponds to the start of the year. Likewise, we specify `-k 3.1` and `-k 3.4` to isolate the month and day portions of the date. We also add the `n` and `r` options to achieve a reverse numeric sort. The `b` option is included to suppress the leading spaces (whose numbers vary from line to line, thereby affecting the outcome of the sort) in the date field.

Some files don't use tabs and spaces as field delimiters; for example, here's the */etc/passwd* file:

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
```

```
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

The fields in this file are delimited with colons (`:`), so how would we sort this file using a key field? `sort` provides the `-t` option to define the field separator character. To sort the *passwd* file on the seventh field (the account's default shell), we could do this:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

By specifying the colon character as the field separator, we can sort on the seventh field.

uniq

Compared to `sort`, the `uniq` program is lightweight. `uniq` performs a seemingly trivial task. When given a sorted file (or standard input), it removes any duplicate lines and sends the results to standard output. It is often used in conjunction with `sort` to clean the output of duplicates.

TIP

> *While `uniq` is a traditional Unix tool often used with `sort`, the GNU version of `sort` supports a `-u` option, which removes duplicates from the sorted output.*

Let's make a text file to try this, as shown here:

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

Remember to type CTRL-D to terminate standard input. Now, if we run `uniq` on our text file, we get this:

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
b
c
```

The results are no different from our original file; the duplicates were not removed. For `uniq` to do its job, the input must be sorted first.

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

This is because `uniq` only removes duplicate lines that are adjacent to each other.

`uniq` has several options. Table 20-2 lists the common ones.

Table 20-2: Common `uniq` Options

Option	Long option	Description
`-c`	`--count`	Output a list of duplicate lines preceded by the number of times the line occurs.
`-d`	`--repeated`	Output only repeated lines, rather than unique lines.
`-f n`	`--skip-fields=n`	Ignore `n` leading fields in each line. Fields are separated by whitespace as they are in `sort`; however, unlike `sort`, `uniq` has no option for setting an alternate field separator.
`-i`	`--ignore-case`	Ignore case during the line comparisons.
`-s n`	`--skip-chars=n`	Skip (ignore) the leading `n` characters of each line.
`-u`	`--unique`	Output only unique lines. Lines with duplicates are ignored.

Here we see `uniq` used to report the number of duplicates found in our text file, using the `-c` option:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
      2 a
```

```
2 b
2 c
```

Slicing and Dicing

The next three programs we will discuss are used to peel columns of text out of files and recombine them in useful ways.

cut—Remove Sections from Each Line of Files

The `cut` program is used to extract a section of text from a line and output the extracted section to standard output. It can accept multiple file arguments or input from standard input.

Specifying the section of the line to be extracted is somewhat awkward and is specified using the options listed in Table 20-3.

Table 20-3: `cut` Selection Options

Option	Long option	Description
`-c` *`list`*	`--characters=`*`list`*	Extract the portion of the line defined by *`list`*. The list may consist of one or more comma-separated numerical ranges.
`-f` *`list`*	`--fields=`*`list`*	Extract one or more fields from the line as defined by *`list`*. The list may contain one or more fields or field ranges separated by commas.
`-d` *`delim`*	`--delimiter=`*`delim`*	When `-f` is specified, use *`delim`* as the field delimiting character. By default, fields must be separated by a single tab character.

Option	Long option	Description
	`--complement`	Extract the entire line of text, except for those portions specified by `-c` and/or `-f`.

As we can see, the way `cut` extracts text is rather inflexible. `cut` is best used to extract text from files that are produced by other programs, rather than text directly typed by humans. We'll take a look at our *distros.txt* file to see whether it is "clean" enough to be a good specimen for our `cut` examples. If we use `cat` with the `-A` option, we can see whether the file meets our requirements of tab-separated fields.

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

It looks good. There are no embedded spaces, just single tab characters between the fields. Because the file uses tabs rather than spaces, we'll use the `-f` option to extract a field.

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Because our *distros* file is tab-delimited, it is best to use `cut` to extract fields rather than characters. This is because when a file is tab-delimited, it is unlikely that each line will contain the same number of characters, which makes calculating character positions within the line difficult or impossible. In our previous example, however, we now have extracted a field that luckily contains data of identical length, so we can show how character extraction works by extracting the year from each line.

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
```

```
2007
2006
2006
2008
2006
2008
2006
```

By running `cut` a second time on our list, we are able to extract character positions 7 through 10, which corresponds to the year in our date field. The `7-10` notation is an example of a range. The `cut` man page contains a complete description of how ranges can be specified.

When working with fields, it is possible to specify a different field delimiter rather than the tab character. Here we will extract the first field from the */etc/passwd* file:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

Using the `-d` option, we are able to specify the colon character as the field delimiter.

EXPANDING TABS

Our *distros.txt* file is ideally formatted for extracting fields using `cut`. But what if we wanted a file that could be fully manipulated with `cut` by char-

acters, rather than fields? This would require us to replace the tab characters within the file with the corresponding number of spaces. Fortunately, the GNU Coreutils package includes a tool for that. Named `expand`, this program accepts either one or more file arguments or standard input and outputs the modified text to standard output.

If we process our *distros.txt* file with `expand`, we can use `cut -c` to extract any range of characters from the file. For example, we could use the following command to extract the year of release from our list by expanding the file and using `cut` to extract every character from the 23rd position to the end of the line:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

Coreutils also provides the `unexpand` program to substitute tabs for spaces.

---

paste—Merge Lines of Files

The `paste` command does the opposite of `cut`. Rather than extracting a column of text from a file, it adds one or more columns of text to a file. It does this by reading multiple files and combining the fields found in each file into a single stream on standard output. Like `cut`, `paste` accepts multiple file arguments and/or standard input. To demonstrate how `paste` operates, we will perform some surgery on our *distros.txt* file to produce a chronological list of releases.

From our earlier work with `sort`, we will first produce a list of distros sorted by date and store the result in a file called *distros-by-date.txt*.

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt >
distros-by-date.txt
```

Next, we will use `cut` to extract the first two fields from the file (the distro name and version) and store that result in a file named *distro-*

versions.txt.

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.txt
[me@linuxbox ~]$ head distros-versions.txt
Fedora    10
Ubuntu    8.10
SUSE      11.0
Fedora    9
Ubuntu    8.04
Fedora    8
Ubuntu    7.10
SUSE      10.3
Fedora    7
Ubuntu    7.04
```

The final piece of preparation is to extract the release dates and store them in a file named *distro-dates.txt*.

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

We now have the parts we need. To complete the process, use `paste` to put the column of dates ahead of the distro names and versions, thus creating a chronological list. This is done simply by using `paste` and ordering its arguments in the desired arrangement.

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008    Fedora     10
10/30/2008    Ubuntu     8.10
06/19/2008    SUSE       11.0
05/13/2008    Fedora     9
04/24/2008    Ubuntu     8.04
11/08/2007    Fedora     8
10/18/2007    Ubuntu     7.10
10/04/2007    SUSE       10.3
05/31/2007    Fedora     7
04/19/2007    Ubuntu     7.04
12/07/2006    SUSE       10.2
10/26/2006    Ubuntu     6.10
10/24/2006    Fedora     6
06/01/2006    Ubuntu     6.06
05/11/2006    SUSE       10.1
03/20/2006    Fedora     5
```

join—Join Lines of Two Files on a Common Field

In some ways, `join` is like `paste` in that it adds columns to a file, but it uses a unique way to do it. A *join* is an operation usually associated with *relational databases* where data from multiple tables with a shared key field is combined to form a desired result. The `join` program performs the same operation. It joins data from multiple files based on a shared key field.

To see how a join operation is used in a relational database, let's imagine a small database consisting of two tables, each containing a single record. The first table, called CUSTOMERS, has three fields: a customer number (CUSTNUM), the customer's first name (FNAME), and the customer's last name (LNAME):

```
CUSTNUM  FNAME LNAME
======== ===== ======
4681934  John  Smith
```

The second table is called ORDERS and contains four fields: an order number (ORDERNUM), the customer number (CUSTNUM), the quantity (QUAN), and the item ordered (ITEM).

```
ORDERNUM      CUSTNUM    QUAN  ITEM
========      =======    ====  ====
3014953305    4681934    1     Blue Widget
```

Note that both tables share the field CUSTNUM. This is important because it allows a relationship between the tables.

Performing a join operation would allow us to combine the fields in the two tables to achieve a useful result, such as preparing an invoice. Using the matching values in the CUSTNUM fields of both tables, a join operation could produce the following:

```
FNAME  LNAME  QUAN   ITEM
=====  =====  ====   ====
John   Smith  1      Blue Widget
```

To demonstrate the `join` program, we'll need to make a couple of files with a shared key. To do this, we will use our *distros-by-date.txt* file. From this file, we will construct two additional files. One contains the release dates (which will be our shared key for this demonstration) and the release names, as shown here:

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008      Fedora
10/30/2008      Ubuntu
06/19/2008      SUSE
05/13/2008      Fedora
04/24/2008      Ubuntu
11/08/2007      Fedora
```

```
10/18/2007      Ubuntu
10/04/2007      SUSE
05/31/2007      Fedora
04/19/2007      Ubuntu
```

The second file contains the release dates and the version numbers, as shown here.

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > dis-
tros-key-vernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008      10
10/30/2008      8.10
06/19/2008      11.0
05/13/2008      9
04/24/2008      8.04
11/08/2007      8
10/18/2007      7.10
10/04/2007      10.3
05/31/2007      7
04/19/2007      7.04
```

We now have two files with a shared key (the “release date” field). It is important to point out that the files must be sorted on the key field for `join` to work properly.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt |
head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
```

10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04

Note also that, by default, `join` uses whitespace as the input field delimiter and a single space as the output field delimiter. This behavior can be modified by specifying options. See the `join` man page for details.

Comparing Text

It is often useful to compare versions of text files. For system administrators and software developers, this is particularly important. A system administrator may, for example, need to compare an existing configuration file to a previous version to diagnose a system problem. Likewise, a programmer frequently needs to see what changes have been made to programs over time.

comm—Compare Two Sorted Files Line by Line

The `comm` program compares two text files and displays the lines that are unique to each one and the lines they have in common. To demonstrate, we will create two nearly identical text files using `cat`.

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

Next, we will compare the two files using `comm`.

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
        b
        c
        d
    e
```

As we can see, `comm` produces three columns of output. The first column contains lines unique to the first file argument, the second column contains the lines unique to the second file argument, and the third column contains the lines shared by both files. `comm` supports options in the form `-n`, where *n* is either 1, 2, or 3. When used, these options specify which columns to suppress. For example, if we wanted to output only the lines shared by both files, we would suppress the output of the first and second columns.

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

diff—Compare Files Line by Line

Like the `comm` program, `diff` is used to detect the differences between files. However, `diff` is a much more complex tool, supporting many output formats and the ability to process large collections of text files at once. `diff` is often used by software developers to examine changes between different versions of program source code and thus has the ability to recursively examine directories of source code, often referred to as *source trees*. One common use for `diff` is the creation of *diff files* or patches that are used by programs such as `patch` (which we'll discuss shortly) to convert one version of a file (or files) to another version.

If we use `diff` to look at our previous example files:

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

we see its default style of output: a terse description of the differences between the two files. In the default format, each group of changes is preceded by a *change command* in the form of *range operation range* to describe the positions and types of changes required to convert the first file to the second file, as outlined in Table 20-4.

Table 20-4: `diff` Change Commands

Change	Description
*r1*a*r2*	Append the lines at the position *r2* in the second file to the position *r1* in the first file.
*r1*c*r2*	Change (replace) the lines at position *r1* with the lines at the position *r2* in the second file.
*r1*d*r2*	Delete the lines in the first file at position *r1*, which would have appeared at range *r2* in the second file

In this format, a range is a comma-separated list of the starting line and the ending line. While this format is the default (mostly for POSIX compliance and backward compatibility with traditional Unix versions of `diff`), it is not as widely used as other, optional formats. Two of the more popular formats are the *context format* and the *unified format*.

When viewed using the context format (the `-c` option), we will see this:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt    2008-12-23 06:40:13.000000000 -0500
```

```
--- file2.txt    2008-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b
  c
  d
+ e
```

The output begins with the names of the two files and their timestamps. The first file is marked with asterisks, and the second file is marked with dashes. Throughout the remainder of the listing, these markers will signify their respective files. Next, we see groups of changes, including the default number of surrounding context lines. In the first group, we see this:

```
*** 1,4 ***
```

which indicates lines 1 through 4 in the first file. Later we see this:

```
--- 1,4 ---
```

which indicates lines 1 through 4 in the second file. Within a change group, lines begin with one of the four indicators described in Table 20-5.

Table 20-5: `diff` Context Format Change Indicators

Indicator	Meaning
`blank`	A line shown for context. It does not indicate a difference between the two files.

Indicator	Meaning
-	A line deleted. This line will appear in the first file but not in the second file.
+	A line added. This line will appear in the second file but not in the first file.
!	A line changed. The two versions of the line will be displayed, each in its respective section of the change group.

The unified format is similar to the context format but is more concise. It is specified with the `-u` option.

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt    2008-12-23 06:40:13.000000000 -0500
+++ file2.txt    2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

The most notable difference between the context and unified formats is the elimination of the duplicated lines of context, making the results of the unified format shorter than those of the context format. In our previous example, we see file timestamps like those of the context format, followed by the string `@@ -1,4 +1,4 @@`. This indicates the lines in the first file and the lines in the second file described in the change group. Following this are the lines themselves, with the default three lines of context. Each line starts with one of the three possible characters described in Table 20-6.

Table 20-6: `diff` Unified Format Change Indicators

Character	Meaning
`blank`	This line is shared by both files.
`-`	This line was removed from the first file.
`+`	This line was added to the first file.

patch—Apply a diff to an Original

The `patch` program is used to apply changes to text files. It accepts output from `diff` and is generally used to convert older-version files into newer versions. Let's consider a famous example. The Linux kernel is developed by a large, loosely organized team of contributors who submit a constant stream of small changes to the source code. The Linux kernel consists of several million lines of code, while the changes that are made by one contributor at one time are quite small. It makes no sense for a contributor to send each developer an entire kernel source tree each time a small change is made. Instead, a diff file is submitted. The diff file contains the change from the previous version of the kernel to the new version with the contributor's changes. The receiver then uses the `patch` program to apply the change to their own source tree. Using `diff`/`patch` offers two significant advantages.

- The diff file is small, compared to the full size of the source tree.
- The diff file concisely shows the change being made, allowing reviewers of the patch to quickly evaluate it.

Of course, `diff`/`patch` will work on any text file, not just source code. It would be equally applicable to configuration files or any other text.

To prepare a diff file for use with `patch`, the GNU documentation suggests using `diff` as follows:

```
diff -Naur old_file new_file > diff_file
```

where *old_file* and *new_file* are either single files or directories containing files. The `r` option supports recursion of a directory tree.

Once the diff file has been created, we can apply it to patch the old file into the new file.

```
patch < diff_file
```

We'll demonstrate with our test file.

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

In this example, we created a diff file named *patchfile.txt* and then used the `patch` program to apply the patch. Note that we did not have to specify a target file to `patch`, as the diff file (in unified format) already contains the filenames in the header. Once the patch is applied, we can see that *file1.txt* now matches *file2.txt*.

`patch` has a large number of options, and there are additional utility programs that can be used to analyze and edit patches.

Editing on the Fly

Our experience with text editors has been largely interactive, meaning that we manually move a cursor around and then type our changes. However, there are *non-interactive* ways to edit text as well. It's possible, for example, to apply a set of changes to multiple files with a single command.

tr—Transliterate or Delete Characters

The `tr` program is used to *transliterate* characters. We can think of this as a sort of character-based search-and-replace operation. Transliteration is the process of changing characters from one alphabet to another. For example, converting characters from lowercase to uppercase is transliteration. We can perform such a conversion with `tr` as follows:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

As we can see, `tr` operates on standard input and outputs its results on standard output. `tr` accepts two arguments: a set of characters to convert from and a corresponding set of characters to convert to. Character sets may be expressed in one of three ways.

- An enumerated list. For example, `ABCDEFGHIJKLMNOPQRSTUVWXYZ`.
- A character range. For example, `A-Z`. Note that this method is sometimes subject to the same issues as other commands because of the locale collation order and thus should be used with caution.
- POSIX character classes. For example, `[:upper:]`.

In most cases, both character sets should be of equal length; however, it is possible for the first set to be larger than the second, particularly if we want to convert multiple characters to a single character.

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

In addition to transliteration, `tr` allows characters to simply be deleted from the input stream. Earlier in this chapter, we discussed the problem of converting MS-DOS text files to Unix-style text. To perform this conversion, carriage return characters need to be removed from the end of each line. This can be performed with `tr` as follows:

```
tr -d '\r' < dos_file > unix_file
```

where *dos_file* is the file to be converted and *unix_file* is the result. This form of the command uses the escape sequence `\r` to represent the carriage return character. To see a complete list of the sequences and character classes `tr` supports, try the following:

```
[me@linuxbox ~]$ tr --help
```

`tr` can perform another trick, too. Using the `-s` option, `tr` can "squeeze" (delete) repeated instances of a character.

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

Here we have a string containing repeated characters. By specifying the set `ab` to `tr`, we eliminate the repeated instances of the letters in the set, while leaving the character that is missing from the set (`c`) unchanged. Note that the repeating characters must be adjoining. If they are not, the squeezing will have no effect.

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

ROT13: THE NOT-SO-SECRET DECODER RING

One amusing use of `tr` is to perform *ROT13 encoding* of text. ROT13 is a trivial type of encryption based on a simple substitution cipher. Calling ROT13 *encryption* is being generous; *text obfuscation* is more accurate. It is used sometimes on text to obscure potentially offensive content. The method simply moves each character 13 places up the alphabet. Because this is halfway up the possible 26 characters, performing the algorithm a second time on the text restores it to its original form. Use the following to perform this encoding with `tr`:

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Performing the same procedure a second time results in the following translation:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

A number of email programs and Usenet news readers support ROT13 encoding. Wikipedia contains a good article on the subject at *http://en.wikipedia.org/wiki/ROT13*.

---

sed—Stream Editor for Filtering and Transforming Text

The name `sed` is short for *stream editor*. It performs text editing on a stream of text, either a set of specified files or standard input. `sed` is a powerful and somewhat complex program (there are entire books about it), so we will not cover it completely here.

In general, the way `sed` works is that it is given either a single editing command (on the command line) or the name of a script file containing multiple commands, and it then performs these commands upon each line in the stream of text. Here is a simple example of `sed` in action:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

In this example, we produce a one-word stream of text using `echo` and pipe it into `sed`. `sed`, in turn, carries out the instruction `s/front/back/` upon the text in the stream and produces the output `back` as a result. We can also recognize this command as resembling the "substitution" (search-and-replace) command in `vi`.

Commands in `sed` begin with a single letter. In the previous example, the substitution command is represented by the letter `s` and is followed by the search-and-replace strings, separated by the slash character as a delimiter. The choice of the delimiter character is arbitrary. By convention,

the slash character is often used, but sed will accept any character that immediately follows the command as the delimiter. We could perform the same command this way:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

By using the underscore character immediately after the command, it becomes the delimiter. The ability to set the delimiter can be used to make commands more readable, as we shall see.

Most commands in sed may be preceded by an *address*, which specifies which line(s) of the input stream will be edited. If the address is omitted, then the editing command is carried out on every line in the input stream. The simplest form of address is a line number. We can add one to our example.

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

Adding the address 1 to our command causes our substitution to be performed on the first line of our one-line input stream. If we specify another number, we see that the editing is not carried out since our input stream does not have a line 2.

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

Addresses may be expressed in many ways. Table 20-7 lists the most common.

Table 20-7: sed Address Notation

Address	Description
n	A line number where *n* is a positive integer.

Address	Description
`$`	The last line.
`/regexp/`	Lines matching a POSIX basic regular expression. Note that the regular expression is delimited by slash characters. Optionally, the regular expression may be delimited by an alternate character, by specifying the expression with `\cregexpc`, where `c` is the alternate character.
`addr1,addr2`	A range of lines from `addr1` to `addr2`, inclusive. Addresses may be any of the single address forms listed earlier.
`first~step`	Match the line represented by the number `first` and then each subsequent line at `step` intervals. For example, `1~2` refers to each odd numbered line, and `5~5` refers to the fifth line and every fifth line thereafter.
`addr1,+n`	Match `addr1` and the following `n` lines.
`addr!`	Match all lines except `addr`, which may be any of the forms listed earlier.

We'll demonstrate different kinds of addresses using the *distros.txt* file from earlier in this chapter. First, here's a range of line numbers:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE        10.2    12/07/2006
Fedora       10     11/25/2008
SUSE        11.0    06/19/2008
Ubuntu       8.04   04/24/2008
Fedora       8      11/08/2007
```

In this example, we print a range of lines, starting with line 1 and continuing to line 5. To do this, we use the `p` command, which simply causes a

matched line to be printed. For this to be effective, however, we must include the option `-n` (the "no auto-print" option) to cause `sed` not to print every line by default.

Next, we'll try a regular expression.

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE        10.2    12/07/2006
SUSE        11.0    06/19/2008
SUSE        10.3    10/04/2007
SUSE        10.1    05/11/2006
```

By including the slash-delimited regular expression `/SUSE/`, we are able to isolate the lines containing it in much the same manner as `grep`.

Finally, we'll try negation by adding an exclamation point (`!`) to the address.

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora      10      11/25/2008
Ubuntu      8.04    04/24/2008
Fedora      8       11/08/2007
Ubuntu      6.10    10/26/2006
Fedora      7       05/31/2007
Ubuntu      7.10    10/18/2007
Ubuntu      7.04    04/19/2007
Fedora      6       10/24/2006
Fedora      9       05/13/2008
Ubuntu      6.06    06/01/2006
Ubuntu      8.10    10/30/2008
Fedora      5       03/20/2006
```

Here we see the expected result: all the lines in the file except the ones matched by the regular expression.

So far, we've looked at two of the `sed` editing commands, `s` and `p`. Table 20-8 provides a more complete list of the basic editing commands.

Table 20-8: `sed` Basic Editing Commands

Command	Description
`=`	Output the current line number.
`a`	Append text after the current line.
`d`	Delete the current line.
`i`	Insert text in front of the current line.
`p`	Print the current line. By default, `sed` prints every line and only edits lines that match a specified address within the file. The default behavior can be overridden by specifying the `-n` option.
`q`	Exit `sed` without processing any more lines. If the `-n` option is not specified, output the current line.
`Q`	Exit `sed` without processing any more lines.

Command	Description
s/*regexp*/*replacement*/	Substitute the contents of *replacement* wherever *regexp* is found. *replacement* may include the special character &, which is equivalent to the text matched by *regexp*. In addition, *replacement* may include the sequences \1 through \9, which are the contents of the corresponding subexpressions in *regexp*. For more about this, see the following discussion of *back references*. After the trailing slash following *replacement*, an optional flag may be specified to modify the s command's behavior.
y/*set1*/*set2*	Perform transliteration by converting characters from *set1* to the corresponding characters in *set2*. Note that unlike tr, sed requires that both sets be of the same length.

The s command is by far the most commonly used editing command. We will demonstrate just some of its power by performing an edit on our *distros.txt* file. We discussed earlier how the date field in *distros.txt* was not in a "computer-friendly" format. While the date is formatted MM/DD/YYYY, it would be better (for ease of sorting) if the format were YYYY-MM-DD. Performing this change on the file by hand would be both time-consuming and error-prone, but with sed, this change can be performed in one step.

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\
)$/\3-\1-\2/' distros.txt
SUSE          10.2     2006-12-07
Fedora         10      2008-11-25
SUSE          11.0     2008-06-19
Ubuntu         8.04     2008-04-24
Fedora         8       2007-11-08
```

```
SUSE        10.3    2007-10-04
Ubuntu      6.10    2006-10-26
Fedora      7       2007-05-31
Ubuntu      7.10    2007-10-18
Ubuntu      7.04    2007-04-19
SUSE        10.1    2006-05-11
Fedora      6       2006-10-24
Fedora      9       2008-05-13
Ubuntu      6.06    2006-06-01
Ubuntu      8.10    2008-10-30
Fedora      5       2006-03-20
```

Wow! Now that is an ugly-looking command. But it works. In just one step, we have changed the date format in our file. It is also a perfect example of why regular expressions are sometimes jokingly referred to as a *write-only* medium. We can write them, but we sometimes cannot read them. Before we are tempted to run away in terror from this command, let's look at how it was constructed. First, we know that the command will have this basic structure:

```
sed 's/regexp/replacement/' distros.txt
```

Our next step is to figure out a regular expression that will isolate the date. Because it is in MM/DD/YYYY format and appears at the end of the line, we can use an expression like this:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

This matches two digits, a slash, two digits, a slash, four digits, and the end of line. So that takes care of `regexp`, but what about `replacement`? To handle that, we must introduce a new regular expression feature that appears in some applications that use BRE. This feature is called *back references* and works like this: if the sequence `\n` appears in `replacement` where `n` is a number from 1 to 9, the sequence will refer to the corresponding subexpression in the preceding regular expression. To create the subexpressions, we simply enclose them in parentheses like so:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

We now have three subexpressions. The first contains the month, the second contains the day of the month, and the third contains the year. Now we can construct *replacement* as follows:

```
\3-\1-\2
```

This gives us the year, a dash, the month, a dash, and the day.

Now, our command looks like this:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

We have two remaining problems. The first is that the extra slashes in our regular expression will confuse `sed` when it tries to interpret the `s` command. The second is that because `sed`, by default, accepts only basic regular expressions, several of the characters in our regular expression will be taken as literals, rather than as metacharacters. We can solve both these problems with a liberal application of backslashes to escape the offending characters.

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' distros.txt
```

And there you have it!

Another feature of the `s` command is the use of optional flags that may follow the replacement string. The most important of these is the `g` flag, which instructs `sed` to apply the search-and-replace globally to a line, not just to the first instance, which is the default. Here is an example:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

We see that the replacement was performed, but only to the first instance of the letter *b*, while the remaining instances were left unchanged. By adding the `g` flag, we are able to change all the instances.

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

So far, we have only given `sed` single commands via the command line. It is also possible to construct more complex commands in a script file using the `-f` option. To demonstrate, we will use `sed` with our *distros.txt* file to build a report. Our report will feature a title at the top, our modified dates, and all the distribution names converted to uppercase. To do this, we will need to write a script, so we'll fire up our text editor and enter the following:

```
# sed script to produce Linux distributions report

1 i\
\
Linux Distributions Report\

s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

We will save our `sed` script as *distros.sed* and run it like this:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE         10.2     2006-12-07
FEDORA       10       2008-11-25
SUSE         11.0     2008-06-19
UBUNTU       8.04     2008-04-24
FEDORA       8        2007-11-08
SUSE         10.3     2007-10-04
UBUNTU       6.10     2006-10-26
FEDORA       7        2007-05-31
UBUNTU       7.10     2007-10-18
UBUNTU       7.04     2007-04-19
```

```
SUSE        10.1    2006-05-11
FEDORA      6       2006-10-24
FEDORA      9       2008-05-13
UBUNTU      6.06    2006-06-01
UBUNTU      8.10    2008-10-30
FEDORA      5       2006-03-20
```

As we can see, our script produces the desired results, but how does it do it? Let's take another look at our script. We'll use `cat` to number the lines.

```
[me@linuxbox ~]$ cat -n distros.sed
  1 # sed script to produce Linux distributions report
  2
  3 1 i\
  4 \
  5 Linux Distributions Report\
  6
  7 s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
  8 y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Line 1 of our script is a *comment*. Like many configuration files and programming languages on Linux systems, comments begin with the `#` character and are followed by human-readable text. Comments can be placed anywhere in the script (though not within commands themselves) and are helpful to any humans who might need to identify and/or maintain the script.

Line 2 is a blank line. Like comments, blank lines may be added to improve readability.

Many `sed` commands support line addresses. These are used to specify which lines of the input are to be acted upon. Line addresses may be expressed as single line numbers, line number ranges, and the special line number `$`, which indicates the last line of input.

Lines 3, 4, 5, and 6 contain text to be inserted at the address 1, the first line of the input. The `i` command is followed by the sequence of a backslash and then a carriage return to produce an escaped carriage return, or what is called a *line-continuation character*. This sequence, which can be used in many circumstances including shell scripts, allows a carriage return to be embedded in a stream of text without signaling the interpreter (in this case `sed`) that the end of the line has been reached. The `i` and the `a` (which appends text, rather than inserting it) and `c` (which replaces text) commands allow multiple lines of text as long as each line, except the last, ends with a line-continuation character. The sixth line of our script is actually the end of our inserted text and ends with a plain carriage return rather than a line-continuation character, signaling the end of the `i` command.

NOTE

A line-continuation character is formed by a backslash followed immediately by a carriage return. No intermediary spaces are permitted.

Line 7 is our search-and-replace command. Since it is not preceded by an address, each line in the input stream is subject to its action.

Line 8 performs transliteration of the lowercase letters into uppercase letters. Note that unlike `tr`, the `y` command in `sed` does not support character ranges (for example, `[a-z]`), nor does it support POSIX. Again, because the `y` command is not preceded by an address, it applies to every line in the input stream.

PEOPLE WHO LIKE SED ALSO LIKE...

`sed` is a capable program, able to perform fairly complex editing tasks to streams of text. It is most often used for simple, one-line tasks rather than long scripts. Many users prefer other tools for larger tasks. The most popular of these are `awk` and `perl`. These go beyond mere tools like the pro-

grams covered here and extend into the realm of complete programming languages. `perl`, in particular, is often used instead of shell scripts for many system management and administration tasks, as well as being a popular medium for web development. `awk` is a little more specialized. Its specific strength is its ability to manipulate tabular data. It resembles `sed` in that `awk` programs normally process text files line by line, using a scheme similar to the `sed` concept of an address followed by an action. While both `awk` and `perl` are outside the scope of this book, they are good skills for the Linux command line user to learn.

---

aspell—Interactive Spellchecker

The last tool we will look at is `aspell`, an interactive spelling checker. The `aspell` program is the successor to an earlier program named `ispell` and can be used, for the most part, as a drop-in replacement. While the `aspell` program is mostly used by other programs that require spellchecking capability, it can also be used effectively as a stand-alone tool from the command line. It has the ability to intelligently check various types of text files, including HTML documents, C or C++ programs, email messages, and other kinds of specialized texts.

To spellcheck a text file containing simple prose, it could be used like this:

```
aspell check textfile
```

where *`textfile`* is the name of the file to check. As a practical example, let's create a simple text file named *foo.txt* containing some deliberate spelling errors.

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

Next we'll check the file using `aspell`.

```
[me@linuxbox ~]$ aspell check foo.txt
```

As `aspell` is interactive in the check mode, we will see a screen like this.

```
The quick brown fox jimped over the laxy dog.

1) jumped                       6) wimped
2) gimped                       7) camped
3) comped                       8) humped
4) limped                       9) impede
5) pimped                       0) umped
i) Ignore                       I) Ignore all
r) Replace                      R) Replace all
a) Add                          l) Add Lower
b) Abort                        x) Exit

?
```

At the top of the display, we see our text with a suspiciously spelled word highlighted. In the middle, we see 10 spelling suggestions numbered 0 through 9, followed by a list of other possible actions. Finally, at the bottom, we see a prompt ready to accept our choice.

If we press the `1` key, `aspell` replaces the offending word with the word *jumped* and moves on to the next misspelled word, which is *laxy*. If we select the replacement *lazy*, `aspell` replaces it and terminates. Once `aspell` has finished, we can examine our file and see that the misspellings have been corrected.

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

Unless told otherwise via the command line option `--dont-backup`, `aspell` creates a backup file containing the original text by appending the extension *.bak* to the filename.

Showing off our `sed` editing prowess, we'll put our spelling mistakes back in so we can reuse our file.

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

The `sed` option `-i` tells `sed` to edit the file "in-place," meaning that rather than sending the edited output to standard output, it will rewrite the file with the changes applied. We also see the ability to place more than one editing command on the line by separating them with a semicolon.

Next, we'll look at how `aspell` can handle different kinds of text files. Using a text editor such as `vim` (the adventurous may want to try `sed`), we will add some HTML markup to our file.

```
<html>
	<head>
		<title>Mispelled HTML file</title>
	</head>
	<body>
		<p>The quick brown fox jimped over the laxy dog.</p>
	</body>
</html>
```

Now, if we try to spellcheck our modified file, we run into a problem. If we do it this way:

```
[me@linuxbox ~]$ aspell check foo.txt
```

we'll get this:

```
< html>
	<head>
		<title>Mispelled HTML file</title>
	</head>
	<body>
		<p>The quick brown fox jimped over the laxy dog.</p>
```

```
    </body>
</html>

1) HTML                 4) Hamel
2) ht ml                5) Hamil
3) ht-ml                6) hotel
i) Ignore               I) Ignore all
r) Replace              R) Replace all
a) Add                  l) Add Lower
b) Abort                x) Exit

?
```

`aspell` will see the contents of the HTML tags as misspelled. This problem can be overcome by including the `-H` (HTML) checking-mode option, like this:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

which will result in this:

```
<html>
    <head>
        <title> Mispelled HTML file</title>
    </head>
    <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
    </body>
</html>


1) Mi spelled           6) Misapplied
2) Mi-spelled           7) Miscalled
3) Misspelled           8) Respelled
4) Dispelled            9) Misspell
5) Spelled              0) Misled
```

```
i) Ignore                I) Ignore all
r) Replace               R) Replace all
a) Add                   l) Add Lower
b) Abort                 x) Exit

?
```

The HTML is ignored, and only the non-markup portions of the file are checked. In this mode, the contents of HTML tags are ignored and not checked for spelling. However, the contents of `ALT` tags, which benefit from checking, are checked in this mode.

NOTE

By default, `aspell` will ignore URLs and email addresses in text. This behavior can be overridden with command line options. It is also possible to specify which markup tags are checked and skipped. See the `aspell` man page for details.

Summing Up

In this chapter, we looked at a few of the many command line tools that operate on text. In the next chapter, we will look at several more. Admittedly, it may not seem immediately obvious how or why you might use some of these tools on a day-to-day basis, though we have tried to show some practical examples of their use. We will find in later chapters that these tools form the basis of a tool set that is used to solve a host of practical problems. This will be particularly true when we get into shell scripting, where these tools will really show their worth.

Extra Credit

There are a few more interesting text-manipulation commands worth investigating. Among these are `split` (split files into pieces), `csplit` (split files into pieces based on context), and `sdiff` (side-by-side merge of file differences).