PART 3

COMMON TASKS AND ESSENTIAL TOOLS

14

PACKAGE MANAGEMENT

If we spend any time in the Linux community, we hear many opinions as to which of the many Linux distributions is "best." Often, these discussions get really silly, focusing on such things as the prettiness of the desktop background (some people won't use Ubuntu because of its default color scheme!) and other trivial matters.

The most important determinant of distribution quality is the *packaging system* and the vitality of the distribution's support community. As we spend more time with Linux, we see that its software landscape is extremely dynamic. Things are constantly changing. Most of the top-tier Linux distributions release new versions every six months and many individual program updates every day. To keep up with this blizzard of software, we need good tools for package management.

Package management is a method of installing and maintaining software on the system. Today, most people can satisfy all of their software needs by installing *packages* from their Linux distributor. This contrasts with the early days of Linux, when one had to download and compile *source code* in order to install software. Not that there is anything wrong with compiling source code; in fact, having access to source code is the great wonder of Linux. It gives us (and everybody else) the ability to examine and improve the system. It's just that working with a precompiled package is faster and easier.

In this chapter, we will look at some of the command-line tools used for package management. While all of the major distributions provide powerful and sophisticated graphical programs for maintaining the system, it is important to learn about the command-line programs, too. They can perform many tasks that are difficult (or impossible) to do using their graphical counterparts.

Packaging Systems

Different distributions use different packaging systems, and as a general rule a package intended for one distribution is not compatible with another distribution. Most distributions fall into one of two camps of packaging technologies: the Debian *.deb* camp and the Red Hat *.rpm* camp. There are some important exceptions, such as Gentoo, Slackware, and Foresight, but most others use one of the two basic systems shown in Table 14-1.

Table 14-1: Major Packaging System Families

Packaging System	Distributions (partial listing)
Debian style (*.deb*)	Debian, Ubuntu, Xandros, Linspire
Red Hat style (*.rpm*)	Fedora, CentOS, Red Hat Enterprise Linux, openSUSE, Mandriva, PCLinuxOS

How a Package System Works

The method of software distribution found in the proprietary software industry usually entails buying a piece of installation media such as an "install disk" and then running an "installation wizard" to install a new application on the system.

Linux doesn't work that way. Virtually all software for a Linux system is found on the Internet. Most of it is provided by the distribution vendor in the form of package files, and the rest is available in source code form, which can be installed manually. We'll talk a little about how to install software by compiling source code in Chapter 23.

Package Files

The basic unit of software in a packaging system is the package file. A *package file* is a compressed collection of files that comprise the software package. A package may consist of numerous programs and data files that support the programs. In addition to the files to be installed, the package file also includes metadata about the package, such as a text description of the package and its contents. Additionally, many packages contain pre- and post-installation scripts that perform configuration tasks before and after the package installation.

Package files are created by a person known as a *package maintainer*, often (but not always) an employee of the distribution vendor. The package maintainer gets the software in source code form from the *upstream provider* (the author of the program), compiles it, and creates the package metadata and any necessary installation scripts. Often, the package maintainer will apply modifications to the original source code to improve the program's integration with the other parts of the Linux distribution.

Repositories

While some software projects choose to perform their own packaging and distribution, most packages today are created by the distribution vendors and interested third parties. Packages are made available to the users of a distribution in central repositories, which may contain many thousands of packages, each specially built and maintained for the distribution.

A distribution may maintain several different repositories for different stages of the software development life cycle. For example, there will usually be a *testing repository*, which contains packages that have just been built and are intended for use by brave souls who are looking for bugs before the packages are released for general distribution. A distribution will often have a *development repository* where work-in-progress packages destined for inclusion in the distribution's next major release are kept.

A distribution may also have related third-party repositories. These are often needed to supply software that, for legal reasons such as patents or Digital Rights Management (DRM) anticircumvention issues, cannot be included with the distribution. Perhaps the best-known case is that of encrypted DVD support, which is not legal in the United States. The third-party repositories operate in countries where software patents and anticircumvention laws do not apply. These repositories are usually wholly independent of the distribution they sup-port, and to use them one must know about them and manually include them in the configuration files for the package management system.

Dependencies

Programs seldom stand alone; rather, they rely on the presence of other software components to get their work done. Common activities, such as input/output for example, are handled by routines shared by many programs. These routines are stored in what are called *shared libraries*, which provide essential services to more than one program. If a package requires a shared resource such as a shared library, it is said to have a *dependency*. Modern package management systems all provide some method of *dependency resolution* to ensure that when a package is installed, all of its dependencies are installed, too.

High- and Low-Level Package Tools

Package management systems usually consist of two types of tools: low-level tools that handle tasks such as installing and removing package files, and high-level tools that perform metadata searching and dependency resolution. In this chapter, we will look at the tools supplied with Debian-style systems (such as Ubuntu and many others) and those used by recent Red Hat products. While all Red Hat–style distributions rely on the same low-level program (`rpm`), they use different high-level tools. For our discussion, we will cover the high-level program `yum`, used by Fedora, Red Hat Enterprise Linux, and CentOS. Other Red Hat–style distributions provide high-level tools with comparable features (see Table 14-2).

Table14-2: Packaging System Tools

Distributions	Low-Level Tools	High-Level Tools
Debian style	`dpkg`	`apt-get`, `aptitude`
Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`yum`

Common Package Management Tasks

Many operations can be performed with the command-line package management tools. We will look at the most common. Be aware that the low-level tools also support creation of package files, an activity outside the scope of this book.

In the following discussion, the term *`package_name`* refers to the actual name of a package, as opposed to *`package_file`*, which is the name of the file that contains the package.

Finding a Package in a Repository

By using the high-level tools to search repository metadata, one can locate a package based on its name or description (see Table 14-3).

Table 14-3: Package Search Commands

Style	Command(s)

Debian	`apt-get update`
	`apt-cache search` *`search_string`*

Red Hat	`yum search` *`search_string`*

Example: Search a yum repository for the emacs text editor on a Red Hat system:

```
yum search emacs
```

Installing a Package from a Repository

High-level tools permit a package to be downloaded from a repository and installed with full dependency resolution (see Table 14-4).

Table 14-4: Package Installation Commands

Style	Command(s)
Debian	`apt-get update` `apt-get install` *package_name*
Red Hat	`yum install` *package_name*

Example: Install the emacs text editor from an apt repository on a Debian-style system:

```
apt-get update; apt-get install emacs
```

Installing a Package from a Package File

If a package file has been downloaded from a source other than a repository, it can be installed directly (though without dependency resolution) using a low-level tool (see Table 14-5).

Table 14-5: Low-Level Package Installation Commands

Style	Command
Debian	`dpkg --install` *package_file*
Red Hat	`rpm -i` *package_file*

Example: If the *emacs-22.1-7.fc7-i386.rpm* package file has been downloaded from a non-repository site, install it on a Red Hat system this way:

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

Note: *Since this technique uses the low-level* `rpm` *program to perform the installation, no dependency resolution is performed. If* `rpm` *discovers a missing dependency,* `rpm` *will exit with an error.*

Removing a Package

Packages can be uninstalled using either the high-level or low-level tools. The high-level tools are shown in Table 14-6.

Table14-6: Package Removal Commands

Style	Command
Debian	`apt-get remove` *package_name*
Red Hat	`yum erase` *package_name*

Example: Uninstall the `emacs` package from a Debian-style system:

```
apt-get remove emacs
```

Updating Packages from a Repository

The most common package management task is keeping the system up-to-date with the latest packages. The high-level tools can perform this vital task in one single step (see Table 14-7).

Table 14-7: Package Update Commands

Style	Command(s)
Debian	`apt-get update; apt-get upgrade`
Red Hat	`yum update`

Example: Apply any available updates to the installed packages on a Debian-style system:

```
apt-get update; apt-get upgrade
```

Upgrading a Package from a Package File

If an updated version of a package has been downloaded from a non-repository source, it can be installed, replacing the previous version (see Table 14-8).

Table 14-8: Low-Level Package Upgrade Commands

Style	Command
Debian	`dpkg --install` *package_file*
Red Hat	`rpm -U` *package_file*

Example: Update an existing installation of emacs to the version contained in the package file *emacs-22.1-7.fc7-i386.rpm* on a Red Hat system:

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

Note: *dpkg does not have a specific option for upgrading a package versus installing one, as rpm does.*

Listing Installed Packages

The commands shown in Table 14-9 can be used to display a list of all the packages installed on the system.

Table 14-9: Package Listing Commands

Style	Command
Debian	`dpkg --list`
Red Hat	`rpm -qa`

Determining Whether a Package Is Installed

The low-level tools shown in Table 14-10 can be used to display whether a specified package is installed.

Table 14-10: Package Status Commands

Style	Command
Debian	`dpkg --status` *package_name*
Red Hat	`rpm -q` *package_name*

Example: Determine whether the emacs package is installed on a Debian-style system:

```
dpkg --status emacs
```

Displaying Information About an Installed Package

If the name of an installed package is known, the commands shown in Table 14-11 can be used to display a description of the package.

Table 14-11: Package Information Commands

Style	Command
Debian	`apt-cache show` *package_name*
Red Hat	`yum info` *package_name*

Example: See a description of the emacs package on a Debian-style system:

```
apt-cache show emacs
```

Finding Which Package Installed a File

To determine which package is responsible for the installation of a particular file, the commands shown in Table 14-12 can be used.

Table 14-12: Package File Identification Commands

Style	Command
Debian	`dpkg --search` *file_name*
Red Hat	`rpm -qf` *file_name*

Example: See which package installed the */usr/bin/vim* file on a Red Hat system:

```
rpm -qf /usr/bin/vim
```

Final Note

In the chapters that follow, we will explore many programs covering a wide range of application areas. While most of these programs are commonly installed by default, sometimes we may need to install additional packages. With our newfound knowledge (and appreciation) of package management, we should have no problem installing and managing the programs we need.

THE LINUX SOFTWARE INSTALLATION MYTH

People migrating from other platforms sometimes fall victim to the myth that software is somehow difficult to install under Linux and that the variety of packaging schemes used by different distributions is a hindrance. Well, it is a hindrance, but only to proprietary software vendors who wish to distribute binary-only versions of their secret software.

The Linux software ecosystem is based on the idea of open source code. If a program developer releases source code for a product, it is likely that a person associated with a distribution will package the product and include it in the repository. This method ensures that the product is well integrated into the distribution and the user is given the convenience of one-stop shopping for software, rather than having to search for each product's website.

Device drivers are handled in much the same way, except that instead of being separate items in a distribution's repository, they become part of the Linux kernel itself. Generally speaking, there is no such thing as a "driver disk" in Linux. Either the kernel supports a device or it doesn't, and the Linux kernel supports a lot of devices. Many more, in fact, than Windows does. Of course, this is no consolation if the particular device you need is not supported. When that happens, you need to look at the cause. A lack of driver support is usually caused by one of three things:

- **The device is too new.** Since many hardware vendors don't actively support Linux development, it falls upon a member of the Linux community to write the kernel driver code. This takes time.
- **The device is too exotic.** Not all distributions include every possible device driver. Each distribution builds its own kernels, and since kernels are very configurable (which is what makes it possible to run Linux on everything from wristwatches to mainframes), the distribution may have overlooked a particular device. By locating and downloading the source code for the driver, it is possible for you (yes, you) to compile and install the driver yourself. This process is not overly difficult, but it is rather involved. We'll talk about compiling software in Chapter 23.
- **The hardware vendor is hiding something.** It has neither released source code for a Linux driver, nor has it released the technical documentation for somebody else to create one. This means that the hardware vendor is trying to keep the programming interfaces to the device a secret. Since we don't want secret devices in our computers, I suggest that you remove the offending hardware and pitch it into the trash with your other useless items.

15

STORAGE MEDIA

In previous chapters we've looked at manipulating data at the file level. In this chapter, we will consider data at the device level. Linux has amazing capabilities for handling storage devices, whether physical storage such as hard disks, network storage, or virtual storage devices like RAID (redundant array of independent disks) and LVM (logical volume manager).

However, since this is not a book about system administration, we will not try to cover this entire topic in depth. What we will do is introduce some of the concepts and key commands that are used to manage storage devices.

To carry out the exercises in this chapter, we will use a USB flash drive, a CD-RW disc (for systems equipped with a CD-ROM burner), and a floppy disk (again, if the system is so equipped).

We will look at the following commands:

- `mount`—Mount a filesystem.
- `umount`—Unmount a filesystem.

- `fdisk`—Partition table manipulator.
- `fsck`—Check and repair a filesystem.
- `fdformat`—Format a floppy disk.
- `mkfs`—Create a filesystem.
- `dd`—Write block-oriented data directly to a device.
- `genisoimage (mkisofs)`—Create an ISO 9660 image file.
- `wodim (cdrecord)`—Write data to optical storage media.
- `md5sum`—Calculate an MD5 checksum.

Mounting and Unmounting Storage Devices

Recent advances in the Linux desktop have made storage device management extremely easy for desktop users. For the most part, we attach a device to our system and it just works. Back in the old days (say, 2004), this stuff had to be done manually. On non-desktop systems (i.e., servers) this is still a largely manual procedure, because servers often have extreme storage needs and complex configuration requirements.

The first step in managing a storage device is attaching the device to the filesystem tree. This process, called *mounting*, allows the device to participate with the operating system. As we recall from Chapter 2, Unix-like operating systems, like Linux, maintain a single filesystem tree with devices attached at various points. This contrasts with other operating systems such as MS-DOS and Windows that maintain separate trees for each device (for example *C:\*, *D:\*, etc.).

A file named */etc/fstab* lists the devices (typically hard disk partitions) that are to be mounted at boot time. Here is an example */etc/fstab* file from a Fedora 7 system:

```
LABEL=/12         /                        ext3    defaults        1 1
LABEL=/home       /home                    ext3    defaults        1 2
LABEL=/boot       /boot                    ext3    defaults        1 2
tmpfs             /dev/shm                 tmpfs   defaults        0 0
devpts            /dev/pts                 devpts  gid=5,mode=620  0 0
sysfs             /sys                     sysfs   defaults        0 0
proc              /proc                    proc    defaults        0 0
LABEL=SWAP-sda3   swap                     swap    defaults        0 0
```

Most of the filesystems listed in this example file are virtual and are not applicable to our discussion. For our purposes, the interesting ones are the first three:

```
LABEL=/12         /                        ext3    defaults        1 1
LABEL=/home       /home                    ext3    defaults        1 2
LABEL=/boot       /boot                    ext3    defaults        1 2
```

These are the hard disk partitions. Each line of the file consists of six fields, as shown in Table 15-1.

Table 15-1: ***/etc/fstab*** **Fields**

Field	Contents	Description
1	Device	Traditionally, this field contains the actual name of a device file associated with the physical device, such as */dev/hda1* (the first partition of the master device on the first IDE channel). But with today's computers, which have many devices that are hot pluggable (like USB drives), many modern Linux distributions associate a device with a text label instead. This label (which is added to the storage medium when it is formatted) is read by the operating system when the device is attached to the system. That way, no matter which device file is assigned to the actual physical device, it can still be correctly identified.
2	Mount point	The directory where the device is attached to the filesystem tree
3	Filesystem type	Linux allows many filesystem types to be mounted. Most native Linux filesystems are ext3, but many others are supported, such as FAT16 (`msdos`), FAT32 (`vfat`), NTFS (`ntfs`), CD-ROM (`iso9660`), etc.
4	Options	Filesystems can be mounted with various options. It is possible, for example, to mount filesystems as read only or to prevent any programs from being executed from them (a useful security feature for removable media).
5	Frequency	A single number that specifies if and when a filesystem is to be backed up with the `dump` command
6	Order	A single number that specifies in what order filesystems should be checked with the `fsck` command

Viewing a List of Mounted Filesystems

The `mount` command is used to mount filesystems. Entering the command without arguments will display a list of the filesystems currently mounted:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
```

```
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext3 (rw)
/dev/sda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime,
uhelper=hal,uid=500,utf8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

The format of the listing is `*device*` `on` `*mount_point*` `type` `*filesystem_type*` (`*options*`). For example, the first line shows that device */dev/sda2* is mounted as the root filesystem, is of type ext3, and is both readable and writable (the option `rw`). This listing also has two interesting entries at the bottom. The next-to-last entry shows a 2-gigabyte SD memory card in a card reader mounted at */media/disk*, and the last entry is a network drive mounted at */misc/musicbox*.

For our first experiment, we will work with a CD-ROM. First, let's look at a system before a CD-ROM is inserted:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

This listing is from a CentOS 5 system that is using LVM to create its root filesystem. Like many modern Linux distributions, this system will attempt to automatically mount the CD-ROM after insertion. After we insert the disc, we see the following:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/hdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,nodev,uid=500)
```

We see the same listing as before, with one additional entry. At the end of the listing, we see that the CD-ROM (which is device */dev/hdc* on this system) has been mounted on */media/live-1.0.10-8* and is type `iso9660` (a CD-ROM). For the purposes of our experiment, we're interested in the name of the device. When you conduct this experiment yourself, the device name will most likely be different.

Warning: *In the examples that follow, it is vitally important that you pay close attention to the actual device names in use on your system and* ***do not use the names used in this text!***

Also, note that audio CDs are not the same as CD-ROMs. Audio CDs do not contain filesystems and thus cannot be mounted in the usual sense.

Now that we have the device name of the CD-ROM drive, let's unmount the disc and remount it at another location in the filesystem tree. To do this, we become the superuser (using the command appropriate for our system) and unmount the disc with the `umount` (notice the spelling) command:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/hdc
```

The next step is to create a new mount point for the disc. A *mount point* is simply a directory somewhere on the filesystem tree. Nothing special about it. It doesn't even have to be an empty directory, though if you mount a device on a non-empty directory, you will not be able to see the directory's previous contents until you unmount the device. For our purposes, we will create a new directory:

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

Finally, we mount the CD-ROM at the new mount point. The `-t` option is used to specify the filesystem type:

```
[root@linuxbox ~]# mount -t iso9660 /dev/hdc /mnt/cdrom
```

Afterward, we can examine the contents of the CD-ROM via the new mount point:

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

Notice what happens when we try to unmount the CD-ROM:

```
[root@linuxbox cdrom]# umount /dev/hdc
umount: /mnt/cdrom: device is busy
```

Why is this? We cannot unmount a device if the device is being used by someone or some process. In this case, we changed our working directory to the mount point for the CD-ROM, which causes the device to be busy. We can easily remedy the issue by changing the working directory to something other than the mount point:

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

Now the device unmounts successfully.

WHY UNMOUNTING IS IMPORTANT

If you look at the output of the `free` command, which displays statistics about memory usage, you will see a statistic called *buffers*. Computer systems are designed to go as fast as possible. One of the impediments to system speed is slow devices. Printers are a good example. Even the fastest printer is extremely slow by computer standards. A computer would be very slow indeed if it had to stop and wait for a printer to finish printing a page. In the early days of PCs (before multitasking), this was a real problem. If you were working on a spreadsheet or text document, the computer would stop and become unavailable every time you printed. The computer would send the data to the printer as fast as the printer could accept it, but it was very slow because printers don't print very fast. This problem was solved by the advent of the *printer buffer*, a device containing some RAM memory, that would sit between the computer and the printer. With the printer buffer in place, the computer would send the printer output to the buffer, and it would quickly be stored in the fast RAM so the computer could go back to work without waiting. Meanwhile, the printer buffer would slowly *spool* the data to the printer from the buffer's memory at the speed at which the printer could accept it.

This idea of buffering is used extensively in computers to make them faster. Don't let the need to occasionally read or write data to or from slow devices impede the speed of the system. Operating systems store data that has been read from, and is to be written to, storage devices in memory for as long as possible before actually having to interact with the slower device. On a Linux system, for example, you will notice that the system seems to fill up memory the longer it is used. This does not mean Linux is "using" all the memory, it means that Linux is taking advantage of all the available memory to do as much buffering as it can.

This buffering allows writing to storage devices to be done very quickly, because the writing to the physical device is being deferred to a future time. In the meantime, the data destined for the device is piling up in memory. From time to time, the operating system will write this data to the physical device.

Unmounting a device entails writing all the remaining data to the device so that it can be safely removed. If the device is removed without first being unmounted, the possibility exists that not all the data destined for the device has been transferred. In some cases, this data may include vital directory updates, which will lead to *filesystem corruption*, one of the worst things that can happen on a computer.

Determining Device Names

It's sometimes difficult to determine the ameof a device. Back in the old days, it wasn't very hard. A device was always in the same place and didn't change. Unix-like systems like it that way. Back when Unix was developed, "changing a disk drive" involved using a forklift to remove a washing machine–sized device from the computer room. In recent years, the typical desktop hardware configuration has become quite dynamic, and Linux has evolved to become more flexible than its ancestors.

In the examples above, we took advantage of the modern Linux desktop's ability to "automagically" mount the device and then determine the name after the fact. But what if we are managing a server or some other environment where this does not occur? How can we figure it out?

First, let's look at how the system names devices. If we list the contents of the */dev* directory (where all devices live), we can see that there are lots and lots of devices:

```
[me@linuxbox ~]$ ls /dev
```

The contents of this listing reveal some patterns of device naming. Table 15-2 lists a few.

Table 15-2: Linux Storage Device Names

Pattern	Device
*/dev/fd**	Floppy disk drives
*/dev/hd**	IDE (PATA) disks on older systems. Typical motherboards contain two IDE connectors, or *channels*, each with a cable with two attachment points for drives. The first drive on the cable is called the *master* device and the second is called the *slave* device. The device names are ordered such that */dev/hda* refers to the master device on the first channel, */dev/hdb* is the slave device on the first channel; */dev/hdc*, the master device on the second channel, and so on. A trailing digit indicates the partition number on the device. For example, */dev/hda1* refers to the first partition on the first hard drive on the system while */dev/hda* refers to the entire drive.
*/dev/lp**	Printers
*/dev/sd**	SCSI disks. On recent Linux systems, the kernel treats all disk-like devices (including PATA/SATA hard disks, flash drives, and USB mass storage devices such as portable music players and digital cameras) as SCSI disks. The rest of the naming system is similar to the older */dev/hd** naming scheme described above.
/dev/sr*	Optical drives (CD/DVD readers and burners)

In addition, we often see symbolic links such as */dev/cdrom*, */dev/dvd*, and */dev/floppy*, which point to the actual device files, provided as a convenience.

If you are working on a system that does not automatically mount removable devices, you can use the following technique to determine how the removable device is named when it is attached. First, start a real-time view of the */var/log/messages* file (you may require superuser privileges for this):

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

The last few lines of the file will be displayed and then pause. Next, plug in the removable device. In this example, we will use a 16MB flash drive. Almost immediately, the kernel will notice the device and probe it:

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device using uhci_h
cd and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass Storage dev
ices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short (5), using
36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy Disk 1.00 PQ:
0 ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware secto
rs (16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write t
hrough
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic sg3 type 0
```

After the display pauses again, press CTRL-C to get the prompt back. The interesting parts of the output are the repeated references to `[sdb]`, which matches our expectation of a SCSI disk device name. Knowing this, two lines become particularly illuminating:

```
Jul 23 10:07:59 linuxbox kernel:  sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

This tells us the device name is */dev/sdb* for the entire device and */dev/sdb1* for the first partition on the device. As we have seen, working with Linux means lots of interesting detective work!

Note: *Using the `tail -f /var/log/messages` technique is a great way to watch what the system is doing in near realtime.*

With our device name in hand, we can now mount the flash drive:

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
```

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5186944   9775164  35% /
/dev/sda5             59631908  31777376  24776480  57% /home
/dev/sda1               147764     17277    122858  13% /boot
tmpfs                   776808         0    776808   0% /dev/shm
/dev/sdb1                15560         0     15560   0% /mnt/flash
```

The device name will remain the same as long as it remains physically attached to the computer and the computer is not rebooted.

Creating New Filesystems

Let's say that we want to reformat the flash drive with a Linux native filesystem, rather than the FAT32 system it has now. This involves two steps: first, (optionally) creating a new partition layout if the existing one is not to our liking, and second, creating a new, empty filesystem on the drive.

Warning: *In the following exercise, we are going to format a flash drive. Use a drive that contains nothing you care about because it will be erased! Again,* ***make absolutely sure you are specifying the correct device name for your system, not the one shown in the text. Failure to heed this warning could result in formatting (i.e., erasing) the wrong drive!***

Manipulating Partitions with fdisk

The `fdisk` program allows us to interact directly with disk-like devices (such as hard disk drives and flash drives) at a very low level. With this tool we can edit, delete, and create partitions on the device. To work with our flash drive, we must first unmount it (if needed) and then invoke the `fdisk` program as follows:

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

Notice that we must specify the device in terms of the entire device, not by partition number. After the program starts up, we will see the following prompt:

```
Command (m for help):
```

Entering an `m` will display the program menu:

```
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)


Command (m for help):
```

The first thing we want to do is examine the existing partition layout. We do this by entering `p` to print the partition table for the device:

```
Command (m for help): p

Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               2        1008       15608+   b  W95 FAT32
```

In this example, we see a 16MB device with a single partition (1) that uses 1006 of the available 1008 cylinders on the device. The partition is identified as a Windows 95 FAT32 partition. Some programs will use this identifier to limit the kinds of operation that can be done to the disk, but most of the time changing the identifier is not critical. However, in the interest of demonstration, we will change it to indicate a Linux partition. To do this, we must first find out what ID is used to identify a Linux partition. In the listing above, we see that the ID `b` is used to specify the existing partition. To see a list of the available partition types, we refer back to the program menu. There we can see the following choice:

```
   l   list known partition types
```

If we enter `l` at the prompt, a large list of possible types is displayed. Among them we see `b` for our existing partition type and `83` for Linux.

Going back to the menu, we see this choice to change a partition ID:

```
   t   change a partition's system id
```

We enter `t` at the prompt and enter the new ID:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

This completes all the changes that we need to make. Up to this point, the device has been untouched (all the changes have been stored in memory, not on the physical device), so we will write the modified partition table to the device and exit.

To do this, we enter w at the prompt:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

If we had decided to leave the device unaltered, we could have entered q at the prompt, which would have exited the program without writing the changes. We can safely ignore the ominous-sounding warning message.

Creating a New Filesystem with mkfs

With our partition editing done (lightweight though it might have been), it's time to create a new filesystem on our flash drive. To do this, we will use mkfs (short for *make filesystem*), which can create filesystems in a variety of formats. To create an ext3 filesystem on the device, we use the -t option to specify the ext3 system type, followed by the name of the device containing the partition we wish to format:

```
[me@linuxbox ~]$ sudo mkfs -t ext3 /dev/sdb1
mke2fs 1.40.2 (12-Jul-2012)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
        8193

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

The program will display a lot of information when ext3 is the chosen filesystem type. To reformat the device to its original FAT32 filesystem, specify vfat as the filesystem type:

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

This process of partitioning and formatting can be used anytime additional storage devices are added to the system. While we worked with a tiny flash drive, the same process can be applied to internal hard disks and other removable storage devices like USB hard drives.

Testing and Repairing Filesystems

In our earlier discussion of the */etc/fstab* file, we saw some mysterious digits at the end of each line. Each time the system boots, it routinely checks the integrity of the filesystems before mounting them. This is done by the `fsck` program (short for *filesystem check*). The last number in each *fstab* entry specifies the order in which the devices are to be checked. In our example above, we see that the root filesystem is checked first, followed by the *home* and *boot* filesystems. Devices with a zero as the last digit are not routinely checked.

In addition to checking the integrity of filesystems, `fsck` can also repair corrupt filesystems with varying degrees of success, depending on the amount of damage. On Unix-like filesystems, recovered portions of files are placed in the *lost+found* directory, located in the root of each filesystem.

To check our flash drive (which should be unmounted first), we could do the following:

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2012)
e2fsck 1.40.8 (13-Mar-2012)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

In my experience, filesystem corruption is quite rare unless there is a hardware problem, such as a failing disk drive. On most systems, filesystem corruption detected at boot time will cause the system to stop and direct you to run `fsck` before continuing.

WHAT THE FSCK?

In Unix culture, fsck is often used in place of a popular word with which it shares three letters. This is especially appropriate, given that you will probably be uttering the aforementioned word if you find yourself in a situation where you are forced to run `fsck`.

Formatting Floppy Disks

For those of us still using computers old enough to be equipped with floppy-disk drives, we can manage those devices, too. Preparing a blank floppy for use is a two-step process. First, we perform a low-level format on the disk, and then we create a filesystem. To accomplish the formatting, we use the `dformat` program specifying the name of the floppy device (usually */dev/fd0*):

```
[me@linuxbox ~]$ sudo fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done
```

Next, we apply a FAT filesystem to the disk with `mkfs`:

```
[me@linuxbox ~]$ sudo mkfs -t msdos /dev/fd0
```

Notice that we use the msdos filesystem type to get the older (and smaller) style file allocation tables. After a disk is prepared, it may be mounted like other devices.

Moving Data Directly to and from Devices

While we usually think of data on our computers as being organized into files, it is also possible to think of the data in "raw" form. If we look at a disk drive, for example, we see that it consists of a large number of "blocks" of data that the operating system sees as directories and files. If we could treat a disk drive as simply a large collection of data blocks, we could perform useful tasks, such as cloning devices.

The `dd` program performs this task. It copies blocks of data from one place to another. It uses a unique syntax (for historical reasons) and is usually used this way:

```
dd if=input_file of=output_file [bs=block_size [count=blocks]]
```

Let's say we had two USB flash drives of the same size and we wanted to exactly copy the first drive to the second. If we attached both drives to the computer and they were assigned to devices */dev/sdb* and */dev/sdc* respectively, we could copy everything on the first drive to the second drive with the following:

```
dd if=/dev/sdb of=/dev/sdc
```

Alternatively, if only the first device were attached to the computer, we could copy its contents to an ordinary file for later restoration or copying:

```
dd if=/dev/sdb of=flash_drive.img
```

Warning: *The* `dd` *command is very powerful. Though its name derives from* data definition, *it is sometimes called* destroy disk *because users often mistype either the* `if` *or* `of` *specifications.* ***Always double-check your input and output specifications before pressing ENTER!***

Creating CD-ROM Images

Writing a recordable CD-ROM (either a CD-R or CD-RW) consists of two steps: first, constructing an *ISO image file* that is the exact filesystem image of the CD-ROM, and second, writing the image file onto the CD-ROM medium.

Creating an Image Copy of a CD-ROM

If we want to make an ISO image of an existing CD-ROM, we can use `dd` to read all the data blocks off the CD-ROM and copy them to a local file. Say we had an Ubuntu CD and we wanted to make an ISO file that we could later use to make more copies. After inserting the CD and determining its device name (we'll assume */dev/cdrom*), we can make the ISO file like so:

```
dd if=/dev/cdrom of=ubuntu.iso
```

This technique works for data DVDs as well, but it will not work for audio CDs as they do not use a filesystem for storage. For audio CDs, look at the `cdrdao` command.

A PROGRAM BY ANY OTHER NAME...

If you look at online tutorials for creating and burning optical media like CD-ROMs and DVDs, you will frequently encounter two programs called `mkisofs` and `cdrecord`. These programs were part of a popular package called `cdrtools` authored by Jörg Schilling. In the summer of 2006, Mr. Schilling made a license change to a portion of the `cdrtools` package that, in the opinion of many in the Linux community, created a license incompatibility with the GNU GPL. As a result, a *fork* of the cdrtools project was started, which now includes replacement programs for `cdrecord` and `mkisofs` named `wodim` and `genisoimage`, respectively.

Creating an Image from a Collection of Files

To create an ISO image file containing the contents of a directory, we use the `enisoimage` program. To do this, we first create a directory containing all the files we wish to include in the image and then execute the `genisoimage` command to create the image file. For example, if we had created a directory called *~/cd-rom-files* and filled it with files for our CD-ROM, we could create an image file named *cd-rom.iso* with the following command:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

The `-R` option adds metadata for the *Rock Ridge extensions*, which allow the use of long filenames and POSIX-style file permissions. Likewise, the `-J` option enables the *Joliet extensions*, which permit long filenames in Windows.

Writing CD-ROM Images

After we have an image file, we can burn it onto our optical media. Most of the commands we discuss below can be applied to both recordable CD-ROM and DVD media.

Mounting an ISO Image Directly

There is a trick that we can use to mount an ISO image while it is still on our hard disk and treat it as though it were already on optical media. By adding the `-o loop` option to `mount` (along with the required `-t iso9660` filesystem type), we can mount the image file as though it were a device and attach it to the filesystem tree:

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

In the example above, we created a mount point named */mnt/iso_image* and then mounted the image file *image.iso* at that mount point. After the image is mounted, it can be treated just as though it were a real CD-ROM or DVD. *Remember to unmount the image when it is no longer needed.*

Blanking a Rewritable CD-ROM

Rewritable CD-RW media need to be erased or *blanked* before being reused. To do this, we can use `wodim`, specifying the device name for the CD writer and the type of blanking to be performed. The `wodim` program offers several types. The most minimal (and fastest) is the `fast` type:

```
wodim dev=/dev/cdrw blank=fast
```

Writing an Image

To write an image, we again use `wodim`, specifying the name of the optical media writer device and the name of the image file:

```
wodim dev=/dev/cdrw image.iso
```

In addition to the device name and image file, `wodim` supports a very large set of options. Two common ones are `-v` for verbose output and `-dao`, which writes the disc in *disc-at-once* mode. This mode should be used if you are preparing a disc for commercial reproduction. The default mode for `wodim` is *track-at-once*, which is useful for recording music tracks.

Extra Credit

It's often useful to verify the integrity of an ISO image that we have downloaded. In most cases, a distributor of an ISO image will also supply a *checksum file*. A checksum is the result of an exotic mathematical calculation resulting in a number that represents the content of the target file. If the contents of the file change by even one bit, the resulting checksum will be much different. The most common method of checksum generation uses the `md5sum` program. When you use `md5sum`, it produces a unique hexadecimal number:

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece  image.iso
```

After you download an image, you should run `md5sum` against it and compare the results with the `md5sum` value supplied by the publisher.

In addition to checking the integrity of a downloaded file, we can use `md5sum` to verify newly written optical media. To do this, we first calculate the checksum of the image file and then calculate a checksum for the medium. The trick to verifying the medium is to limit the calculation to only the portion of the optical medium that contains the image. We do this by determining the number of 2048-byte blocks the image contains (optical media is always written in 2048-byte blocks) and reading that many blocks from the medium. On some types of media, this is not required. A CD-R written in disc-at-once mode can be checked this way:

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece  /dev/cdrom
```

Many types of media, such as DVDs, require a precise calculation of the number of blocks. In the example below, we check the integrity of the image file *dvd-image.iso* and the disc in the DVD reader */dev/dvd*. Can you figure out how this works?

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s" dvd-image
.iso) / 2048 )) | md5sum
```

16

NETWORKING

When it comes to networking, there is probably nothing that cannot be done with Linux. Linux is used to build all sorts of networking systems and appliances, including firewalls, routers, name servers, NAS (network-attached storage) boxes, and on and on.

Just as the subject of networking is vast, so is the number of commands that can be used to configure and control it. We will focus our attention on just a few of the most frequently used ones. The commands chosen for examination include those used to monitor networks and those used to transfer files. In addition, we are going to explore the `ssh` program, which is used to perform remote logins. This chapter will cover the following:

- `ping`—Send an ICMP ECHO_REQUEST to network hosts.
- `traceroute`—Print the route packets trace to a network host.
- `netstat`—Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.
- `ftp`—Internet file transfer program.

- `lftp`—An improved Internet file transfer program.
- `wget`—Non-interactive network downloader.
- `ssh`—OpenSSH SSH client (remote login program).
- `scp`—Secure copy (remote file copy program).
- `sftp`—Secure file transfer program.

We're going to assume a little background in networking. In this, the Internet age, everyone using a computer needs a basic understanding of networking concepts. To make full use of this chapter, you should be familiar with the following terms:

- IP (Internet protocol) address
- Host and domain name
- URI (uniform resource identifier)

Note: *Some of the commands we will cover may (depending on your distribution) require the installation of additional packages from your distribution's repositories, and some may require superuser privileges to execute.*

Examining and Monitoring a Network

Even if you're not the system administrator, it's often helpful to examine the performance and operation of a network.

ping—Send a Special Packet to a Network Host

The most basic network command is `ping`. The `ping` command sends a special network packet called an IMCP ECHO_REQUEST to a specified host. Most network devices receiving this packet will reply to it, allowing the network connection to be verified.

Note: *It is possible to configure most network devices (including Linux hosts) to ignore these packets. This is usually done for security reasons, to partially obscure a host from a potential attacker. It is also common for firewalls to be configured to block IMCP traffic.*

For example, to see if we can reach *http://www.linuxcommand.org/* (one of my favorite sites ;-)), we can use `ping` like this:

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Once started, `ping` continues to send packets at a specified interval (default is 1 second) until it is interrupted:

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
```

```
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1 ttl=43 time=10
7 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2 ttl=43 time=10
8 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3 ttl=43 time=10
6 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4 ttl=43 time=10
6 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5 ttl=43 time=10
5 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6 ttl=43 time=10
7 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

After it is interrupted (in this case after the sixth packet) by the pressing of CTRL-C, `ping` prints performance statistics. A properly performing network will exhibit zero percent packet loss. A successful ping will indicate that the elements of the network (its interface cards, cabling, routing, and gateways) are in generally good working order.

traceroute—Trace the Path of a Network Packet

The `traceroute` program (some systems use the similar `tracepath` program instead) displays a listing of all the "hops" network traffic takes to get from the local system to a specified host. For example, to see the route taken to reach *http://www.slashdot.org/*, we would do this:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

The output looks like this:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte packets
 1  ipcop.localdomain (192.168.1.1)  1.066 ms  1.366 ms  1.720 ms
 2  * * *
 3  ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9)  14.622 ms  14.885
ms  15.169 ms
 4  po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154)  17.634 ms  17.626
ms  17.899 ms
 5  po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158)  15.992 ms  15.983
ms  16.256 ms
 6  po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5)  22.835 ms  14.23
3 ms  14.405 ms
 7  po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34)  16.154 ms  13.600
ms  18.867 ms
 8  te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77)  21.951 ms
21.073 ms  21.557 ms
 9  pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10)  22.917 ms  21
.884 ms  22.126 ms
10  204.70.144.1 (204.70.144.1)  43.110 ms  21.248 ms  21.264 ms
11  cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93)  21.857 ms cr2-pos-0-0-
3-1.newyork.savvis.net (204.70.204.238)  19.556 ms cr1-pos-0-7-3-1.newyork.sav
vis.net (204.70.195.93)  19.634 ms
```

```
12  cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109)  41.586 ms  42.843 ms
cr2-tengig-0-0-2-0.chicago.savvis.net (204.70.196.242)  43.115 ms
13  hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net (204.70.195.122)  44.21
5 ms  41.833 ms  45.658 ms
14  csr1-ve241.elkgrovech3.savvis.net (216.64.194.42)  46.840 ms  43.372 ms  4
7.041 ms
15  64.27.160.194 (64.27.160.194)  56.137 ms  55.887 ms  52.810 ms
16  slashdot.org (216.34.181.45)  42.727 ms  42.016 ms  41.437 ms
```

In the output, we can see that connecting from our test system to *http://www.slashdot.org/* requires traversing 16 routers. For routers that provide identifying information, we see their hostnames, IP addresses, and performance data, which include three samples of round-trip time from the local system to the router. For routers that do not provide identifying information (because of router configuration, network congestion, firewalls, etc.), we see asterisks as in the line for hop number two.

netstat—Examine Network Settings and Statistics

The `netstat` program is used to examine various network settings and statistics. Through the use of its many options, we can look at a variety of features in our network setup. Using the `-ie` option, we can examine the network interfaces in our system:

```
[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet  HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:153098921 (146.0 MB)  TX bytes:261035246 (248.9 MB)
        Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:111490 (108.8 KB)  TX bytes:111490 (108.8 KB)
```

In the example above, we see that our test system has two network interfaces. The first, called `eth0`, is the Ethernet interface; the second, called `lo`, is the *loopback interface*, a virtual interface that the system uses to "talk to itself."

When performing causal network diagnostics, the important things to look for are the presence of the word `UP` at the beginning of the fourth line for each interface, indicating that the network interface is enabled, and the presence of a valid IP address in the `inet addr` field on the second line. For systems using Dynamic Host Configuration Protocol (DHCP), a valid IP address in this field will verify that the DHCP is working.

Using the `-r` option will display the kernel's network routing table. This shows how the network is configured to send packets from network to network:

```
[me@linuxbox ~]$ netstat -r
Kernel IP routing table
Destination  Gateway     Genmask       Flags   MSS Window  irtt Iface
192.168.1.0  *           255.255.255.0 U         0 0          0 eth0  default
192.168.1.1 0.0.0.0       UG        0 0          0 eth0
```

In this simple example, we see a typical routing table for a client machine on a local area network (LAN) behind a firewall/router. The first line of the listing shows the destination `192.168.1.0`. IP addresses that end in zero refer to networks rather than individual hosts, so this destination means any host on the LAN. The next field, `Gateway`, is the name or IP address of the gateway (router) used to go from the current host to the destination network. An asterisk in this field indicates that no gateway is needed.

The last line contains the destination `default`. This means any traffic destined for a network that is not otherwise listed in the table. In our example, we see that the gateway is defined as a router with the address of `192.168.1.1`, which presumably knows what to do with the destination traffic.

The `netstat` program has many options, and we have looked at only a couple. Check out the `netstat` man page for a complete list.

Transporting Files over a Network

What good is a network unless we know how to move files across it? There are many programs that move data over networks. We will cover two of them now and several more in later sections.

ftp—Transfer Files with the File Transfer Protocol

One of the true "classic" programs, `ftp` gets its name from the protocol it uses, the *File Transfer Protocol*. FTP is used widely on the Internet for file downloads. Most, if not all, web browsers support it, and you often see URIs starting with the protocol *ftp://*.

Before there were web browsers, there was the `ftp` program. `ftp` is used to communicate with *FTP servers*, machines that contain files that can be uploaded and downloaded over a network.

FTP (in its original form) is not secure, because it sends account names and passwords in *cleartext*. This means that they are not encrypted and anyone sniffing the network can see them. Because of this, almost all FTP done over the Internet is done by *anonymous FTP servers*. An anonymous server allows anyone to log in using the login name *anonymous* and a meaningless password.

In the following example, we show a typical session with the `ftp` program downloading an Ubuntu ISO image located in the */pub/cd_images/Ubuntu-8.04* directory of the anonymous FTP server *fileserver*.

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-8.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-rw-r--    1 500      500      733079552 Apr 25 03:53 ubuntu-8.04-desktop-
i386.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-8.04-desktop-i386.iso
local: ubuntu-8.04-desktop-i386.iso remote: ubuntu-8.04-desktop-i386.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-8.04-desktop-i386.iso
(733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

Table 16-1 gives an explanation of the commands entered during this session.

Table 16-1: Examples of Interactive ftp Commands

Command	Meaning
`ftp fileserver`	Invoke the `ftp` program and have it connect to the FTP server *fileserver*.
`anonymous`	Login name. After the login prompt, a password prompt will appear. Some servers will accept a blank password. Others will require a password in the form of an email address. In that case, try something like *user@example.com*.
`cd pub/cd_images/Ubuntu-8.04`	Change to the directory on the remote system containing the desired file. Note that on most anonymous FTP servers, the files for public downloading are found somewhere under the *pub* directory.
`ls`	List the directory on the remote system.

Table 16-1 (*continued*)

Command	Meaning
`lcd Desktop`	Change the directory on the local system to *~/Desktop*. In the example, the `ftp` program was invoked when the working directory was ~. This command changes the working directory to *~/Desktop*.
`get ubuntu-8.04-desktop-i386.iso`	Tell the remote system to transfer the file *ubuntu-8.04-desktop-i386.iso* to the local system. Since the working directory on the local system was changed to *~/Desktop*, the file will be downloaded there.
`bye`	Log off the remote server and end the `ftp` program session. The commands `quit` and `exit` may also be used.

Typing `help` at the `ftp>` prompt will display a list of the supported commands. Using `ftp` on a server where sufficient permissions have been granted, it is possible to perform many ordinary file management tasks. It's clumsy, but it does work.

lftp—A Better ftp

`ftp` is not the only command-line FTP client. In fact, there are many. One of the better (and more popular) ones is `lftp` by Alexander Lukyanov. It works much like the traditional `ftp` program but has many additional convenience features, including multiple-protocol support (including HTTP), automatic retry on failed downloads, background processes, tab completion of pathnames, and many more.

wget—Non-interactive Network Downloader

Another popular command-line program for file downloading is `wget`. It is useful for downloading content from both web and FTP sites. Single files, multiple files, and even entire sites can be downloaded. To download the first page of *http://www.linuxcommand.org/*, we could do this:

```
[me@linuxbox ~]$  wget http://linuxcommand.org/index.php
--11:02:51--  http://linuxcommand.org/index.php
           => `index.php'
Resolving linuxcommand.org... 66.35.250.210
Connecting to linuxcommand.org|66.35.250.210|:80... connected.
```

```
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

    [ <=>                                ] 3,120         --.--K/s

11:02:51 (161.75 MB/s) - `index.php' saved [3120]
```

The program's many options allow `wget` to recursively download, download files in the background (allowing you to log off but continue downloading), and complete the download of a partially downloaded file. These features are well documented in its better-than-average man page.

Secure Communication with Remote Hosts

For many years, Unix-like operating systems have had the ability to be administered remotely via a network. In the early days, before the general adoption of the Internet, there were a couple of popular programs used to log in to remote hosts: the `rlogin` and `telnet` programs. These programs, however, suffer from the same fatal flaw that the `ftp` program does; they transmit all their communications (including login names and passwords) in cleartext. This makes them wholly inappropriate for use in the Internet age.

ssh—Securely Log in to Remote Computers

To address this problem, a new protocol called SSH (Secure Shell) was developed. SSH solves the two basic problems of secure communication with a remote host. First, it authenticates that the remote host is who it says it is (thus preventing man-in-the-middle attacks), and second, it encrypts all of the communications between the local and remote hosts.

SSH consists of two parts. An SSH server runs on the remote host, listening for incoming connections on port 22, while an SSH client is used on the local system to communicate with the remote server.

Most Linux distributions ship an implementation of SSH called `OpenSSH` from the BSD project. Some distributions include both the client and the server packages by default (for example, Red Hat), while others (such as Ubuntu) supply only the client. To enable a system to receive remote connections, it must have the `OpenSSH-server` package installed, configured, and running, and (if the system is either running or behind a firewall) it must allow incoming network connections on TCP port 22.

Note: *If you don't have a remote system to connect to but want to try these examples, make sure the* `OpenSSH-server` *package is installed on your system and use* `localhost` *as the name of the remote host. That way, your machine will create network connections with itself.*

The SSH client program used to connect to remote SSH servers is called, appropriately enough, ssh. To connect to a remote host named remote-sys, we would use the ssh client program like so:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be established.
RSA key fingerprint is 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?
```

The first time the connection is attempted, a message is displayed indicating that the authenticity of the remote host cannot be established. This is because the client program has never seen this remote host before. To accept the credentials of the remote host, enter yes when prompted. Once the connection is established, the user is prompted for a password:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list of known
hosts.
me@remote-sys's password:
```

After the password is successfully entered, we receive the shell prompt from the remote system:

```
Last login: Tue Aug 30 13:00:48 2011
[me@remote-sys ~]$
```

The remote shell session continues until the user enters the exit command at the remote shell prompt, thereby closing the remote connection. At this point, the local shell session resumes, and the local shell prompt reappears.

It is also possible to connect to remote systems using a different username. For example, if the local user *me* had an account named *bob* on a remote system, user *me* could log in to the account *bob* on the remote system as follows:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Tue Aug 30 13:03:21 2011
[bob@remote-sys ~]$
```

As stated before, ssh verifies the authenticity of the remote host. If the remote host does not successfully authenticate, the following message appears:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
```

```
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested strict
checking.
Host key verification failed.
```

This message is caused by one of two possible situations. First, an attacker may be attempting a man-in-the-middle attack. This is rare, because everybody knows that ssh alerts the user to this. The more likely culprit is that the remote system has been changed somehow; for example, its operating system or SSH server has been reinstalled. In the interests of security and safety, however, the first possibility should not be dismissed out of hand. Always check with the administrator of the remote system when this message occurs.

After determining that the message is due to a benign cause, it is safe to correct the problem on the client side. This is done by using a text editor (vim perhaps) to remove the obsolete key from the *~/.ssh/known_hosts* file. In the example message above, we see this:

```
Offending key in /home/me/.ssh/known_hosts:1
```

This means that line 1 of the *known_hosts* file contains the offending key. Delete this line from the file, and the ssh program will be able to accept new authentication credentials from the remote system.

Besides opening a shell session on a remote system, ssh also allows us to execute a single command on a remote system. For example, we can execute the free command on a remote host named *remote-sys* and have the results displayed on the local system:

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
             total       used       free     shared    buffers     cached
Mem:        775536     507184     268352          0     110068     154596
-/+ buffers/cache:     242520     533016
Swap:      1572856          0    1572856
[me@linuxbox ~]$
```

It’s possible to use this technique in more interesting ways, such as this example in which we perform an ls on the remote system and redirect the output to a file on the local system:

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Notice the use of the single quotes. This is done because we do not want the pathname expansion performed on the local machine; rather, we want it to be performed on the remote system. Likewise, if we had wanted the output redirected to a file on the remote machine, we could have placed the redirection operator and the filename within the single quotes:

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

TUNNELING WITH SSH

Part of what happens when you establish a connection with a remote host via SSH is that an *encrypted tunnel* is created between the local and remote systems. Normally, this tunnel is used to allow commands typed at the local system to be transmitted safely to the remote system and the results to be transmitted safely back. In addition to this basic function, the SSH protocol allows most types of network traffic to be sent through the encrypted tunnel, creating a sort of *VPN* (virtual private network) between the local and remote systems.

Perhaps the most common use of this feature is to allow X Window system traffic to be transmitted. On a system running an X server (that is, a machine displaying a GUI), it is possible to launch and run an X client program (a graphical application) on a remote system and have its display appear on the local system. It's easy to do—here's an example. Let's say we are sitting at a Linux system called *linuxbox* that is running an X server, and we want to run the xload program on a remote system named *remote-sys* and see the program's graphical output on our local system. We could do this:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 05 13:23:11 2011
[me@remote-sys ~]$ xload
```

After the xload command is executed on the remote system, its window appears on the local system. On some systems, you may need to use the -Y option rather than the -X option to do this.

scp and sftp—Securely Transfer Files

The OpenSSH package also includes two programs that can make use of an SSH-encrypted tunnel to copy files across the network. The first, scp (secure copy) is used much like the familiar cp program to copy files. The most notable difference is that the source or destination pathname may be preceded with the name of a remote host followed by a colon character. For example, if we wanted to copy a document named *document.txt* from our home directory on the remote system, *remote-sys*, to the current working directory on our local system, we could do this:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                              100% 5581     5.5KB/s   00:00
[me@linuxbox ~]$
```

As with ssh, you may apply a username to the beginning of the remote host's name if the desired remote host account name does not match that of the local system:

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

The second SSH file-copying program is sftp, which, as its name implies, is a secure replacement for the ftp program. sftp works much like the original ftp program that we used earlier; however, instead of transmitting everything in cleartext, it uses an SSH-encrypted tunnel. sftp has an important advantage over conventional ftp in that it does not require an FTP server to be running on the remote host. It requires only the SSH server. This means that any remote machine that can connect with the SSH client can also be used as a FTP-like server. Here is a sample session:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-desktop-i386.iso

/home/me/ubuntu-8.04-desktop-i386.iso 100%  699MB   7.4MB/s   01:35
sftp> bye
```

Note: *The SFTP protocol is supported by many of the graphical file managers found in Linux distributions. Using either Nautilus (GNOME) or Konqueror (KDE), we can enter a URI beginning with* sftp:// *into the location bar and operate on files stored on a remote system running an SSH server.*

AN SSH CLIENT FOR WINDOWS?

Let's say you are sitting at a Windows machine but you need to log in to your Linux server and get some real work done. What do you do? Get an SSH client program for your Windows box, of course! There are a number of these. The most popular one is probably PuTTY by Simon Tatham and his team. The PuTTY program displays a terminal window and allows a Windows user to open an SSH (or telnet) session on a remote host. The program also provides analogs for the scp and sftp programs.

PuTTY is available at *http://www.chiark.greenend.org.uk/~sgtatham/putty/*.

17

SEARCHING FOR FILES

As we have wandered around our Linux system, one thing has become abundantly clear: A typical Linux system has a lot of files! This raises the question "How do we find things?" We already know that the Linux filesystem is well organized according to conventions that have been passed down from one generation of Unix-like systems to the next, but the sheer number of files can present a daunting problem.

In this chapter, we will look at two tools that are used to find files on a system:

- `locate`—Find files by name.
- `find`—Search for files in a directory hierarchy.

We will also look at a command that is often used with file-search commands to process the resulting list of files:

- `xargs`—Build and execute command lines from standard input.

In addition, we will introduce a couple of commands to assist us in our explorations:

- `touch`—Change file times.
- `stat`—Display file or filesystem status.

locate—Find Files the Easy Way

The `locate` program performs a rapid database search of pathnames and then outputs every name that matches a given substring. Say, for example, we want to find all the programs with names that begin with *zip*. Since we are looking for programs, we can assume that the name of the directory containing the programs would end with *bin/*. Therefore, we could try to use `locate` this way to find our files:

```
[me@linuxbox ~]$ locate bin/zip
```

`locate` will search its database of pathnames and output any that contain the string `bin/zip`:

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

If the search requirement is not so simple, `locate` can be combined with other tools, such as `grep`, to design more interesting searches:

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

The `locate` program has been around for a number of years, and several different variants are in common use. The two most common ones found in modern Linux distributions are `slocate` and `mlocate`, though they are usually accessed by a symbolic link named locate. The different versions of locate have overlapping options sets. Some versions include regular-expression matching (which we'll cover in Chapter 19) and wildcard support. Check the man page for locate to determine which version of locate is installed.

WHERE DOES THE LOCATE DATABASE COME FROM?

You may notice that, on some distributions, locate fails to work just after the system is installed, but if you try again the next day, it works fine. What gives? The locate database is created by another program named updatedb. Usually, it is run periodically as a *cron job*; that is, a task performed at regular intervals by the cron daemon. Most systems equipped with locate run updatedb once a day. Since the database is not updated continuously, you will notice that very recent files do not show up when using locate. To overcome this, it's possible to run the updatedb program manually by becoming the superuser and running updatedb at the prompt.

find—Find Files the Hard Way

While the locate program can find a file based solely on its name, the find program searches a given directory (and its subdirectories) for files based on a variety of attributes. We're going to spend a lot of time with find because it has a bunch of interesting features that we will see again and again when we start to cover programming concepts in later chapters.

In its simplest use, find is given one or more names of directories to search. For example, it can produce a list of our home directory:

```
[me@linuxbox ~]$ find ~
```

On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let's use wc to count the number of files:

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

Wow, we've been busy! The beauty of find is that it can be used to identify files that meet specific criteria. It does this through the (slightly strange) application of *tests*, *actions*, and *options*. We'll look at the tests first.

Tests

Let's say that we want a list of directories from our search. To do this, we could add the following test:

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

Adding the test `-type d` limited the search to directories. Conversely, we could have limited the search to regular files with this test:

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

Table 17-1 lists the common file-type tests supported by `find`.

Table 17-1: find File Types

File Type	Description
`b`	Block special device file
`c`	Character special device file
`d`	Directory
`f`	Regular file
`l`	Symbolic link

We can also search by file size and filename by adding some additional tests. Let's look for all the regular files that match the wildcard pattern **.JPG* and are larger than 1 megabyte:

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

In this example, we add the `-name` test followed by the wildcard pattern. Notice that we enclose it in quotes to prevent pathname expansion by the shell. Next, we add the `-size` test followed by the string `+1M`. The leading plus sign indicates that we are looking for files larger than the specified number. A leading minus sign would change the string to mean "smaller than the specified number." Using no sign means "match the value exactly." The trailing letter `M` indicates that the unit of measurement is megabytes. The characters shown in Table 17-2 may be used to specify units.

Table 17-2: find Size Units

Character	Unit
`b`	512-byte blocks (the default if no unit is specified)
`c`	Bytes
`w`	2-byte words
`k`	Kilobytes (units of 1024 bytes)
`M`	Megabytes (units of 1,048,576 bytes)
`G`	Gigabytes (units of 1,073,741,824 bytes)

`find` supports a large number of different tests. Table 17-3 provides a rundown of the common ones. Note that in cases where a numeric argument is required, the same + and - notation discussed above can be applied.

Table 17-3: find Tests

Test	Description
`-cmin` *`n`*	Match files or directories whose content or attributes were last modified exactly *n* minutes ago. To specify fewer than *n* minutes ago, use -*n*; to specify more than *n* minutes ago, use +*n*.
`-cnewer` *`file`*	Match files or directories whose contents or attributes were last modified more recently than those of *`file`*.
`-ctime` *`n`*	Match files or directories whose contents or attributes (i.e., permissions) were last modified *n**24 hours ago.
`-empty`	Match empty files and directories.
`-group` *`name`*	Match file or directories belonging to group *`name`*. *`name`* may be expressed as either a group name or as a numeric group ID.
`-iname` *`pattern`*	Like the `-name` test but case insensitive.
`-inum` *`n`*	Match files with inode number *n*. This is helpful for finding all the hard links to a particular inode.
`-mmin` *`n`*	Match files or directories whose contents were modified *n* minutes ago.
`-mtime` *`n`*	Match files or directories whose contents only were last modified *n**24 hours ago.
`-name` *`pattern`*	Match files and directories with the specified wildcard *`pattern`*.
`-newer` *`file`*	Match files and directories whose contents were modified more recently than the specified *`file`*. This is very useful when writing shell scripts that perform file backups. Each time you make a backup, update a file (such as a log) and then use `find` to determine which files have changed since the last update.
`-nouser`	Match file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
`-nogroup`	Match files and directories that do not belong to a valid group.

(*continued*)

Table 17-3 (*continued*)

Test	Description
`-perm` *`mode`*	Match files or directories that have permissions set to the specified *`mode`*. *`mode`* may be expressed by either octal or symbolic notation.
`-samefile` *`name`*	Similar to the `-inum` test. Matches files that share the same inode number as file *`name`*.
`-size` *`n`*	Match files of size *`n`*.
`-type` *`c`*	Match files of type *`c`*.
`-user` *`name`*	Match files or directories belonging to *`name`*. *`name`* may be expressed by a username or by a numeric user ID.

This is not a complete list. The `find` man page has all the details.

Operators

Even with all the tests that `find` provides, we may still need a better way to describe the *logical relationships* between the tests. For example, what if we needed to determine if all the files and subdirectories in a directory had secure permissions? We would look for all the files with permissions that are not 0600 and the directories with permissions that are not 0700. Fortunately, `find` provides a way to combine tests using *logical operators* to create more complex logical relationships. To express the aforementioned test, we could do this:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d -not -perm
0700 \)
```

Yikes! That sure looks weird. What is all this stuff? Actually, the operators are not that complicated once you get to know them (see Table 17-4).

Table 17-4: find Logical Operators

Operator	Description
`-and`	Match if the tests on both sides of the operator are true. May be shortened to `-a`. Note that when no operator is present, `-and` is implied by default.
`-or`	Match if a test on either side of the operator is true. May be shortened to `-o`.
`-not`	Match if the test following the operator is false. May be shortened to `-!`.

Table 17-4 (*continued*)

Operator	Description
( )	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, `find` evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve readability of the command. Note that since the parentheses characters have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to `find`. Usually the backslash character is used to escape them.

With this list of operators in hand, let's deconstruct our `find` command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an `-or` operator:

```
(expression 1) -or (expression 2)
```

This makes sense, since we are searching for files with a certain set of permissions and for directories with a different set. If we are looking for both files and directories, why do we use `-or` instead of `-and`? Because as `find` scans through the files and directories, each one is evaluated to see if it matches the specified tests. We want to know if it is *either* a file with bad permissions *or* a directory with bad permissions. It can't be both at the same time. So if we expand the grouped expressions, we can see it this way:

```
(file with bad perms) -or (directory with bad perms)
```

Our next challenge is how to test for "bad permissions." How do we do that? Actually we don't. What we will test for is "not good permissions," since we know what "good permissions" are. In the case of files, we define *good* as 0600; for directories, 0700. The expression that will test files for "not good" permissions is:

```
-type f -and -not -perms 0600
```

and the expression for directories is:

```
-type d -and -not -perms 0700
```

As noted in Table 17-4, the `-and` operator can be safely removed, since it is implied by default. So if we put this all back together, we get our final command:

```
find ~ (-type f -not -perms 0600) -or (-type d -not -perms 0700)
```

However, since the parentheses have special meaning to the shell, we must escape them to prevent the shell from trying to interpret them. Preceding each one with a backslash character does the trick.

There is another feature of logical operators that is important to understand. Let's say that we have two expressions separated by a logical operator:

```
expr1 -operator expr2
```

In all cases, *expr1* will always be performed; however, the operator will determine if *expr2* is performed. Table 17-5 shows how it works.

Table 17-5: find AND/OR Logic

Results of *expr1*	Operator	*expr2* is...
True	`-and`	Always performed
False	`-and`	Never performed
True	`-or`	Never performed
False	`-or`	Always performed

Why does this happen? It's done to improve performance. Take `-and`, for example. We know that the expression *expr1* `-and` *expr2* cannot be true if the result of *expr1* is false, so there is no point in performing *expr2*. Likewise, if we have the expression *expr1* `-or` *expr2* and the result of *expr1* is true, there is no point in performing *expr2*, as we already know that the expression *expr1* `-or` *expr2* is true.

Okay, so this helps things go faster. Why is this important? Because we can rely on this behavior to control how actions are performed, as we shall soon see.

Actions

Let's get some work done! Having a list of results from our `find` command is useful, but what we really want to do is act on the items on the list. Fortunately, `find` allows actions to be performed based on the search results.

Predefined Actions

There are a set of predefined actions and several ways to apply user-defined actions. First let's look at a few of the predefined actions in Table 17-6.

Table 17-6: Predefined find Actions

Action	Description
`-delete`	Delete the currently matching file.
`-ls`	Perform the equivalent of `ls -dils` on the matching file. Output is sent to standard output.
`-print`	Output the full pathname of the matching file to standard output. This is the default action if no other action is specified.

Table 17-6 (*continued*)

Action	Description
`-quit`	Quit once a match has been made.

As with the tests, there are many more actions. See the `find` man page for full details.

In our very first example, we did this:

```
find ~
```

This command produced a list of every file and subdirectory contained within our home directory. It produced a list because the `-print` action is implied if no other action is specified. Thus, our command could also be expressed as

```
find ~ -print
```

We can use `find` to delete files that meet certain criteria. For example, to delete files that have the file extension *.BAK* (which is often used to designate backup files), we could use this command:

```
find ~ -type f -name '*.BAK' -delete
```

In this example, every file in the user's home directory (and its subdirectories) is searched for filenames ending in *.BAK*. When they are found, they are deleted.

Warning: *It should go without saying that you should* ***use extreme caution*** *when using the* `-delete` *action. Always test the command first by substituting the* `-print` *action for* `-delete` *to confirm the search results.*

Before we go on, let's take another look at how the logical operators affect actions. Consider the following command:

```
find ~ -type f -name '*.BAK' -print
```

As we have seen, this command will look for every regular file (`-type f`) whose name ends with *.BAK* (`-name '*.BAK'`) and will output the relative pathname of each matching file to standard output (`-print`). However, the reason the command performs the way it does is determined by the logical relationships between each of the tests and actions. Remember, there is, by default, an implied `-and` relationship between each test and action. We could also express the command this way to make the logical relationships easier to see:

```
find ~ -type f -and -name '*.BAK' -and -print
```

With our command fully expressed, let's look at Table 17-7 to see how the logical operators affect its execution.

Table 17-7: Effect of Logical Operators

Test/Action	Is performed when...
`-print`	`-type f` and `-name '*.BAK'` are true.
`-name '*.BAK'`	`-type f` is true.
`-type f`	Is always performed, since it is the first test/action in an `-and` relationship.

Since the logical relationship between the tests and actions determines which of them are performed, we can see that the order of the tests and actions is important. For instance, if we were to reorder the tests and actions so that the `-print` action was the first one, the command would behave much differently:

```
find ~ -print -and -type f -and -name '*.BAK'
```

This version of the command will print each file (the `-print` action always evaluates to true) and then test for file type and the specified file extension.

User-Defined Actions

In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the `-exec` action, like this:

`-exec` *`command`* `{} ;`

where *command* is the name of a command, `{}` is a symbolic representation of the current pathname, and the semicolon is a required delimiter indicating the end of the command. Here's an example of using `-exec` to act like the `-delete` action discussed earlier:

```
-exec rm '{}' ';'
```

Again, since the brace and semicolon characters have special meaning to the shell, they must be quoted or escaped.

It's also possible to execute a user-defined action interactively. By using the `-ok` action in place of `-exec`, the user is prompted before execution of each specified command:

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

In this example, we search for files with names starting with the string `foo` and execute the command `ls -l` each time one is found. Using the `-ok` action prompts the user before the `ls` command is executed.

Improving Efficiency

When the -exec action is used, it launches a new instance of the specified command each time a matching file is found. There are times when we might prefer to combine all of the search results and launch a single instance of the command. For example, rather than executing the commands like this,

```
ls -l file1
ls -l file2
```

we may prefer to execute them this way:

```
ls -l file1 file2
```

Here we cause the command to be executed only one time rather than multiple times. There are two ways we can do this: the traditional way, using the external command xargs, and the alternative way, using a new feature in find itself. We'll talk about the alternative way first.

By changing the trailing semicolon character to a plus sign, we activate the ability of find to combine the results of the search into an argument list for a single execution of the desired command. Going back to our example,

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

will execute ls each time a matching file is found. By changing the command to

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

we get the same results, but the system has to execute the ls command only once.

We can also use the xargs command to get the same result. xargs accepts input from standard input and converts it into an argument list for a specified command. With our example, we would use it like this:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me   me 224 2011-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2012-09-19 12:53 /home/me/foo.txt
```

Here we see the output of the find command piped into xargs, which, in turn, constructs an argument list for the ls command and then executes it.

Note: *While the number of arguments that can be placed into a command line is quite large, it's not unlimited. It is possible to create commands that are too long for the shell to accept. When a command line exceeds the maximum length supported by the system,* `xargs` *executes the specified command with the maximum number of arguments possible and then repeats this process until standard input is exhausted. To see the maximum size of the command line, execute* `xargs` *with the* `--show-limits` *option.*

DEALING WITH FUNNY FILENAMES

Unix-like systems allow embedded spaces (and even newlines!) in filenames. This causes problems for programs like xargs that construct argument lists for other programs. An embedded space will be treated as a delimiter, and the resulting command will interpret each space-separated word as a separate argument. To overcome this, find and xarg allow the optional use of a *null character* as argument separator. A null character is defined in ASCII as the character represented by the number zero (as opposed to, for example, the space character, which is defined in ASCII as the character represented by the number 32). The find command provides the action -print0, which produces null-separated output, and the xargs command has the --null option, which accepts null separated input. Here's an example:

```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Using this technique, we can ensure that all files, even those containing embedded spaces in their names, are handled correctly.

A Return to the Playground

It's time to put find to some (almost) practical use. First, let's create a playground with lots of subdirectories and files:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

Marvel in the power of the command line! With these two lines, we created a playground directory containing 100 subdirectories, each containing 26 empty files. Try that with the GUI!

The method we employed to accomplish this magic involved a familiar command (mkdir); an exotic shell expansion (braces); and a new command, touch. By combining mkdir with the -p option (which causes mkdir to create the parent directories of the specified paths) with brace expansion, we were able to create 100 directories.

The touch command is usually used to set or update the modification times of files. However, if a filename argument is that of a non-existent file, an empty file is created.

In our playground, we created 100 instances of a file named *file-A*. Let's find them:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

Note that unlike ls, find does not produce results in sorted order. Its order is determined by the layout of the storage device. We can confirm that we actually have 100 instances of the file this way:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

Next, let's look at finding files based on their modification times. This will be helpful when creating backups or organizing files in chronological order. To do this, we will first create a reference file against which we will compare modification time:

```
[me@linuxbox ~]$ touch playground/timestamp
```

This creates an empty file named `timestamp` and sets its modification time to the current time. We can verify this by using another handy command, `stat`, which is a kind of souped-up version of `ls`. The `stat` command reveals all that the system understands about a file and its attributes:

```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2012-10-08 15:15:39.000000000 -0400
Modify: 2012-10-08 15:15:39.000000000 -0400
Change: 2012-10-08 15:15:39.000000000 -0400
```

If we `touch` the file again and then examine it with `stat`, we will see that the file's times have been updated:

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2012-10-08 15:23:33.000000000 -0400
Modify: 2012-10-08 15:23:33.000000000 -0400
Change: 2012-10-08 15:23:33.000000000 -0400
```

Next, let's use `find` to update some of our playground files:

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch '{}' ';'
```

This updates all files in the playground that are named *file-B*. Next we'll use `find` to identify the updated files by comparing all the files to the reference file *timestamp*:

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

The results contain all 100 instances of *file-B*. Since we performed a `touch` on all the files in the playground that are named *file-B* after we updated *timestamp*, they are now "newer" than *timestamp* and thus can be identified with the `-newer` test.

Finally, let's go back to the bad permissions test we performed earlier and apply it to *playground*:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \( -type d
-not -perm 0700 \)
```

This command lists all 100 directories and 2,600 files in *playground* (as well as *timestamp* and *playground* itself, for a total of 2,702) because none of them meets our definition of "good permissions." With our knowledge of operators and actions, we can add actions to this command to apply new permissions to the files and directories in our playground:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600
'{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700 '{}' ';' \)
```

On a day-to-day basis, we might find it easier to issue two commands, one for the directories and one for the files, rather than this one large compound command, but it's nice to know that we can do it this way. The important point here is to understand how operators and actions can be used together to perform useful tasks.

Options

Finally, we have the options. The options are used to control the scope of a `find` search. They may be included with other tests and actions when constructing `find` expressions. Table 17-8 lists the most commonly used options.

Table 17-8: find Options

Option	Description
`-depth`	Direct `find` to process a directory's files before the directory itself. This option is automatically applied when the `-delete` action is specified.
`-maxdepth` *`levels`*	Set the maximum number of levels that `find` will descend into a directory tree when performing tests and actions.
`-mindepth` *`levels`*	Set the minimum number of levels that `find` will descend into a directory tree before applying tests and actions.
`-mount`	Direct `find` not to traverse directories that are mounted on other filesystems.
`-noleaf`	Direct `find` not to optimize its search based on the assumption that it is searching a Unix-like filesystem. This is needed when scanning DOS/Windows filesystems and CD-ROMs.

18

ARCHIVING AND BACKUP

One of the primary tasks of a computer system's administrator is to keep the system's data secure. One way this is done is by performing timely backups of the system's files. Even if you're not a system administrator, it is often useful to make copies of things and to move large collections of files from place to place and from device to device.

In this chapter, we will look at several common programs that are used to manage collections of files. There are the file compression programs:

- `gzip`—Compress or expand files.
- `bzip2`—A block sorting file compressor.

the archiving programs:

- `tar`—Tape-archiving utility.
- `zip`—Package and compress files.

and the file synchronization program:

- `rsync`—Remote file and directory synchronization.

Compressing Files

Throughout the history of computing, there has been a struggle to get the most data into the smallest available space, whether that space be memory, storage devices, or network bandwidth. Many of the data services that we take for granted today, such as portable music players, high-definition television, or broadband Internet, owe their existence to effective *data compression* techniques.

Data compression is the process of removing *redundancy* from data. Let's consider an imaginary example. Say we had an entirely black picture file with the dimensions of 100 pixels by 100 pixels. In terms of data storage (assuming 24 bits, or 3 bytes per pixel), the image will occupy 30,000 bytes of storage: $100 \times 100 \times 3 = 30{,}000$.

An image that is all one color contains entirely redundant data. If we were clever, we could encode the data in such a way as to simply describe the fact that we have a block of 30,000 black pixels. So, instead of storing a block of data containing 30,000 zeros (black is usually represented in image files as zero), we could compress the data into the number 30,000, followed by a zero to represent our data. Such a data compression scheme, called *run-length encoding*, is one of the most rudimentary compression techniques. Today's techniques are much more advanced and complex, but the basic goal remains the same—get rid of redundant data.

Compression algorithms (the mathematical techniques used to carry out the compression) fall into two general categories, *lossless* and *lossy*. Lossless compression preserves all the data contained in the original. This means that when a file is restored from a compressed version, the restored file is exactly the same as the original, uncompressed version. Lossy compression, on the other hand, removes data as the compression is performed, to allow more compression to be applied. When a lossy file is restored, it does not match the original version; rather, it is a close approximation. Examples of lossy compression are JPEG (for images) and MP3 (for music). In our discussion, we will look exclusively at lossless compression, since most data on computers cannot tolerate any data loss.

gzip—Compress or Expand Files

The `gzip` program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding `gunzip` program is used to restore compressed files to their original, uncompressed form. Here is an example:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
```

```
-rw-r--r-- 1 me    me    15738 2012-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    3230 2012-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2012-10-14 07:15 foo.txt
```

In this example, we create a text file named *foo.txt* from a directory listing. Next, we run `gzip`, which replaces the original file with a compressed version named *foo.txt.gz*. In the directory listing of *foo.**, we see that the original file has been replaced with the compressed version and that the compressed version is about one-fifth the size of the original. We can also see that the compressed file has the same permissions and time stamp as the original.

Next, we run the `gunzip` program to uncompress the file. Afterward, we can see that the compressed version of the file has been replaced with the original, again with the permissions and timestamp preserved.

`gzip` has many options. Table 18-1 lists a few.

Table 18-1: gzip Options

Option	Description
`-c`	Write output to standard output and keep original files. May also be specified with `--stdout` and `--to-stdout`.
`-d`	Decompress. This causes `gzip` to act like `gunzip`. May also be specified with `--decompress` or `--uncompress`.
`-f`	Force compression even if a compressed version of the original file already exists. May also be specified with `--force`.
`-h`	Display usage information. May also be specified with `--help`.
`-l`	List compression statistics for each file compressed. May also be specified with `--list`.
`-r`	If one or more arguments on the command line are directories, recursively compress files contained within them. May also be specified with `--recursive`.
`-t`	Test the integrity of a compressed file. May also be specified with `--test`.
`-v`	Display verbose messages while compressing. May also be specified with `--verbose`.
`-`*`number`*	Set amount of compression. *number* is an integer in the range of 1 (fastest, least compression) to 9 (slowest, most compression). The values 1 and 9 may also be expressed as `--fast` and `--best`, respectively. The default value is 6.

Let's look again at our earlier example:

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:      OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

Here, we replaced the file *foo.txt* with a compressed version named *foo.txt.gz*. Next, we tested the integrity of the compressed version, using the -t and -v options. Finally, we decompressed the file back to its original form.

gzip can also be used in interesting ways via standard input and output:

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

This command creates a compressed version of a directory listing.

The gunzip program, which uncompresses gzip files, assumes that filenames end in the extension *.gz*, so it's not necessary to specify it, as long as the specified name is not in conflict with an existing uncompressed file:

```
[me@linuxbox ~]$ gunzip foo.txt
```

If our goal were only to view the contents of a compressed text file, we could do this:

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

Alternatively, a program supplied with gzip, called zcat, is equivalent to gunzip with the -c option. It can be used like the cat command on gzip-compressed files:

```
[me@linuxbox ~]$ zcat foo.txt.gz | less
```

Note: *There is a* zless *program, too. It performs the same function as the pipeline above.*

bzip2—Higher Compression at the Cost of Speed

The bzip2 program, by Julian Seward, is similar to gzip but uses a different compression algorithm, which achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as gzip. A file compressed with bzip2 is denoted with the extension *.bz2*:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me    me    15738 2012-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me    me     2792 2012-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

As we can see, `bzip2` can be used the same way as `gzip`. All the options (except for `-r`) that we discussed for `gzip` are also supported in `bzip2`. Note, however, that the compression level option (`-`*`number`*) has a somewhat different meaning to `bzip2`. `bzip2` comes with `bunzip2` and `bzcat` for decompressing files.

`bzip2` also comes with the `bzip2recover` program, which will try to recover damaged *.bz2* files.

DON'T BE COMPRESSIVE COMPULSIVE

I occasionally see people attempting to compress a file that has already been compressed with an effective compression algorithm, by doing something like this:

```
$ gzip picture.jpg
```

Don't do it. You're probably just wasting time and space! If you apply compression to a file that is already compressed, you will actually end up with a larger file. This is because all compression techniques involve some overhead that is added to the file to describe the compression. If you try to compress a file that already contains no redundant information, the compression will not result in any savings to offset the additional overhead.

Archiving Files

A common file-management task used in conjunction with compression is *archiving*. Archiving is the process of gathering up many files and bundling them into a single large file. Archiving is often done as a part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

tar—Tape Archiving Utility

In the Unix-like world of software, the `tar` program is the classic tool for archiving files. Its name, short for *tape archive*, reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept on other storage devices. We often see filenames that end with the extension *.tar* or *.tgz*, which indicate a "plain" tar archive and a gzipped archive, respectively. A tar archive can consist of a group of separate files, one or more directory hierarchies, or a mixture of both. The command syntax works like this:

```
tar mode[options] pathname...
```

where *`mode`* is one of the operating modes shown in Table 18-2 (only a partial list is shown here; see the `tar` man page for a complete list).

Table 18-2: tar Modes

Mode	Description
c	Create an archive from a list of files and/or directories.
x	Extract an archive.
r	Append specified pathnames to the end of an archive.
t	List the contents of an archive.

tar uses a slightly odd way of expressing options, so we'll need some examples to show how it works. First, let's re-create our playground from the previous chapter:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{00{1..9},0{10..99},100}
[me@linuxbox ~]$ touch playground/dir-{00{1..9},0{10..99},100}/file-{A..Z}
```

Next, let's create a tar archive of the entire playground:

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

This command creates a tar archive named *playground.tar*, which contains the entire playground directory hierarchy. We can see that the mode and the f option, which is used to specify the name of the tar archive, may be joined together and do not require a leading dash. Note, however, that the mode must always be specified first, before any other option.

To list the contents of the archive, we can do this:

```
[me@linuxbox ~]$ tar tf playground.tar
```

For a more detailed listing, we can add the v (verbose) option:

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Now, let's extract the playground in a new location. We will do this by creating a new directory named *foo*, changing the directory, and extracting the tar archive:

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

If we examine the contents of *~/foo/playground*, we see that the archive was successfully installed, creating a precise reproduction of the original files. There is one caveat, however: Unless you are operating as the superuser, files and directories extracted from archives take on the ownership of the user performing the restoration, rather than the original owner.

Another interesting behavior of `tar` is the way it handles pathnames in archives. The default for pathnames is relative, rather than absolute. `tar` does this by simply removing any leading slash from the pathname when creating the archive. To demonstrate, we will re-create our archive, this time specifying an absolute pathname:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Remember, *~/playground* will expand into */home/me/playground* when we press the ENTER key, so we will get an absolute pathname for our demonstration. Next, we will extract the archive as before and watch what happens:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
[me@linuxbox foo]$ ls
home   playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

Here we can see that when we extracted our second archive, it re-created the directory *home/me/playground* relative to our current working directory, *~/foo*, not relative to the root directory, as would have been the case with an absolute pathname. This may seem like an odd way for it to work, but it's actually more useful this way, as it allows us to extract archives to any location rather than being forced to extract them to their original locations. Repeating the exercise with the inclusion of the verbose option (`v`) will give a clearer picture of what's going on.

Let's consider a hypothetical, yet practical, example of `tar` in action. Imagine we want to copy the home directory and its contents from one system to another and we have a large USB hard drive that we can use for the transfer. On our modern Linux system, the drive is "automagically" mounted in the */media* directory. Let's also imagine that the disk has a volume name of *BigDisk* when we attach it. To make the tar archive, we can do the following:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

After the tar file is written, we unmount the drive and attach it to the second computer. Again, it is mounted at */media/BigDisk*. To extract the archive, we do this:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

What's important to see here is that we must first change directory to / so that the extraction is relative to the root directory, since all pathnames within the archive are relative.

When extracting an archive, it's possible to limit what is extracted. For example, if we wanted to extract a single file from an archive, it could be done like this:

```
tar xf archive.tar pathname
```

By adding the trailing *pathname* to the command, we ensure that `tar` will restore only the specified file. Multiple pathnames may be specified. Note that the pathname must be the full, exact relative pathname as stored in the archive. When specifying pathnames, wildcards are not normally supported; however, the GNU version of `tar` (which is the version most often found in Linux distributions) supports them with the `--wildcards` option. Here is an example using our previous *playground.tar* file:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/playground/
dir-*/file-A'
```

This command will extract only files matching the specified pathname including the wildcard *dir-**.

`tar` is often used in conjunction with `find` to produce archives. In this example, we will use `find` to produce a set of files to include in an archive:

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf playground.tar '{
}' '+'
```

Here we use `find` to match all the files in *playground* named *file-A* and then, using the `-exec` action, we invoke `tar` in the append mode (`r`) to add the matching files to the archive *playground.tar*.

Using `tar` with `find` is a good way to create *incremental backups* of a directory tree or an entire system. By using `find` to match files newer than a timestamp file, we could create an archive that contains only files newer than the last archive, assuming that the timestamp file is updated right after each archive is created.

`tar` can also make use of both standard input and output. Here is a comprehensive example:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-from=- | gzip
> playground.tgz
```

In this example, we used the `find` program to produce a list of matching files and piped them into `tar`. If the filename `-` is specified, it is taken to mean standard input or output, as needed. (By the way, this convention of using `-` to represent standard input/output is used by a number of other programs, too.) The `--files-from` option (which may also be specified as `-T`) causes `tar` to read its list of pathnames from a file rather than the command line. Lastly, the archive produced by `tar` is piped into `gzip` to create the compressed archive *playground.tgz*. The *.tgz* extension is the conventional extension given to `gzip`-compressed tar files. The extension *.tar.gz* is also used sometimes.

While we used the gzip program externally to produce our compressed archive, modern versions of GNU tar support both gzip and bzip2 compression directly with the use of the z and j options, respectively. Using our previous example as a base, we can simplify it this way:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf playground.tgz -T -
```

If we had wanted to create a bzip2-compressed archive instead, we could have done this:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf playground.tbz -T -
```

By simply changing the compression option from z to j (and changing the output file's extension to *.tbz* to indicate a bzip2-compressed file), we enabled bzip2 compression.

Another interesting use of standard input and output with the tar command involves transferring files between systems over a network. Imagine that we had two machines running a Unix-like system equipped with tar and ssh. In such a scenario, we could transfer a directory from a remote system (named remote-sys for this example) to our local system:

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar xf -
me@remote-sys's password:
[me@linuxbox remote-stuff]$ ls
Documents
```

Here we were able to copy a directory named *Documents* from the remote system *remote-sys* to a directory within the directory named *remote-stuff* on the local system. How did we do this? First, we launched the tar program on the remote system using ssh. You will recall that ssh allows us to execute a program remotely on a networked computer and "see" the results on the local system—the standard output produced on the remote system is sent to the local system for viewing. We can take advantage of this by having tar create an archive (the c mode) and send it to standard output, rather than a file (the f option with the dash argument), thereby transporting the archive over the encrypted tunnel provided by ssh to the local system. On the local system, we execute tar and have it expand an archive (the x mode) supplied from standard input (again, the f option with the dash argument).

zip—Package and Compress Files

The zip program is both a compression tool and an archiver. The file format used by the program is familiar to Windows users, as it reads and writes *.zip* files. In Linux, however, gzip is the predominant compression program with bzip2 being a close second. Linux users mainly use zip for exchanging files with Windows systems, rather than performing compression and archiving.

In its most basic usage, zip is invoked like this:

```
zip options zipfile file...
```

For example, to make a zip archive of our playground, we would do this:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

Unless we include the -r option for recursion, only the *playground* directory (but none of its contents) is stored. Although the addition of the extension *.zip* is automatic, we will include the file extension for clarity.

During the creation of the zip archive, zip will normally display a series of messages like this:

```
 adding: playground/dir-020/file-Z (stored 0%)
 adding: playground/dir-020/file-Y (stored 0%)
 adding: playground/dir-020/file-X (stored 0%)
 adding: playground/dir-087/ (stored 0%)
 adding: playground/dir-087/file-S (stored 0%)
```

These messages show the status of each file added to the archive. zip will add files to the archive using one of two storage methods: Either it will "store" a file without compression, as shown here, or it will "deflate" the file, which performs compression. The numeric value displayed after the storage method indicates the amount of compression achieved. Since our playground contains only empty files, no compression is performed on its contents.

Extracting the contents of a zip file is straightforward when using the unzip program:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

One thing to note about zip (as opposed to tar) is that if an existing archive is specified, it is updated rather than replaced. This means that the existing archive is preserved, but new files are added and matching files are replaced.

Files may be listed and extracted selectively from a zip archive by specifying them to unzip:

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive:  ./playground.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
        0  10-05-12 09:25   playground/dir-087/file-Z
 --------                   -------
        0                   1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-Z
Archive:  ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
 extracting: playground/dir-087/file-Z
```

Using the -l option causes unzip to merely list the contents of the archive without extracting the file. If no file(s) are specified, unzip will list all files in the archive. The -v option can be added to increase the verbosity of the listing. Note that when the archive extraction conflicts with an existing file, the user is prompted before the file is replaced.

Like tar, zip can make use of standard input and output, though its implementation is somewhat less useful. It is possible to pipe a list of filenames to zip via the -@ option:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

Here we use find to generate a list of files matching the test -name "file-A" and then pipe the list into zip, which creates the archive *file-A.zip* containing the selected files.

zip also supports writing its output to standard output, but its use is limited because very few programs can make use of the output. Unfortunately, the unzip program does not accept standard input. This prevents zip and unzip from being used together to perform network file copying like tar.

zip can, however, accept standard input, so it can be used to compress the output of other programs:

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
 adding: - (deflated 80%)
```

In this example, we pipe the output of ls into zip. Like tar, zip interprets the trailing dash as "use standard input for the input file."

The unzip program allows its output to be sent to standard output when the -p (for pipe) option is specified:

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

We touched on some of the basic things that zip and unzip can do. They both have a lot of options that add to their flexibility, though some are platform specific to other systems. The man pages for both zip and unzip are pretty good and contain useful examples.

Synchronizing Files and Directories

A common strategy for maintaining a backup copy of a system involves keeping one or more directories synchronized with another directory (or directories) located on either the local system (usually a removable storage device of some kind) or a remote system. We might, for example, have a local copy of a website under development and synchronize it from time to time with the "live" copy on a remote web server.

rsync—Remote File and Directory Synchronization

In the Unix-like world, the preferred tool for this task is `rsync`. This program can synchronize both local and remote directories by using the *rsync remote-update protocol*, which allows `rsync` to quickly detect the differences between two directories and perform the minimum amount of copying required to bring them into sync. This makes `rsync` very fast and economical to use, compared to other kinds of copy programs.

`rsync` is invoked like this:

```
rsync options source destination
```

where *`source`* and *`destination`* are each one of the following:

- A local file or directory
- A remote file or directory in the form of *[user@]host:path*
- A remote rsync server specified with a URI of *rsync://[user@]host[:port]/path*

Note that either the source or the destination must be a local file. Remote-to-remote copying is not supported.

Let's try `rsync` out on some local files. First, let's clean out our *foo* directory:

```
[me@linuxbox ~]$ rm -rf foo/*
```

Next, we'll synchronize the *playground* directory with a corresponding copy in *foo*:

```
[me@linuxbox ~]$ rsync -av playground foo
```

We've included both the `-a` option (for archiving—causes recursion and preservation of file attributes) and the `-v` option (verbose output) to make a *mirror* of the *playground* directory within *foo*. While the command runs, we will see a list of the files and directories being copied. At the end, we will see a summary message like this, indicating the amount of copying performed:

```
sent 135759 bytes  received 57870 bytes  387258.00 bytes/sec
total size is 3230  speedup is 0.02
```

If we run the command again, we will see a different result:

```
[me@linuxbox ~]$ rsync -av playgound foo
building file list ... done

 sent 22635 bytes  received 20 bytes  45310.00 bytes/sec
total size is 3230  speedup is 0.14
```

Notice that there was no listing of files. This is because `rsync` detected that there were no differences between *~/playground* and *~/foo/playground*, and therefore it didn't need to copy anything. If we modify a file in *playground* and run `rsync` again, we see that `rsync` detected the change and copied only the updated file.

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
playground/dir-099/file-Z
sent 22685 bytes  received 42 bytes  45454.00 bytes/sec
total size is 3230  speedup is 0.14
```

As a practical example, let's consider the imaginary external hard drive that we used earlier with tar. If we attach the drive to our system and, once again, it is mounted at */media/BigDisk*, we can perform a useful system backup by first creating a directory named */backup* on the external drive and then using rsync to copy the most important stuff from our system to the external drive:

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/
backup
```

In this example, we copied the */etc*, */home*, and */usr/local* directories from our system to our imaginary storage device. We included the --delete option to remove files that may have existed on the backup device that no longer existed on the source device (this is irrelevant the first time we make a backup but will be useful on subsequent copies). Repeating the procedure of attaching the external drive and running this rsync command would be a useful (though not ideal) way of keeping a small system backed up. Of course, an alias would be helpful here, too. We could create an alias and add it to our *.bashrc* file to provide this feature:

```
alias backup='sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/bac
kup'
```

Now all we have to do is attach our external drive and run the backup command to do the job.

Using rsync over a Network

One of the real beauties of rsync is that it can be used to copy files over a network. After all, the *r* in rsync stands for *remote*. Remote copying can be done in one of two ways.

The first way is with another system that has rsync installed, along with a remote shell program such as ssh. Let's say we had another system on our local network with a lot of available hard drive space and we wanted to perform our backup operation using the remote system instead of an external drive. Assuming that it already had a directory named */backup* where we could deliver our files, we could do this:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home /usr/local remote-
sys:/backup
```

We made two changes to our command to facilitate the network copy. First, we added the `--rsh=ssh` option, which instructs `rsync` to use the `ssh` program as its remote shell. In this way, we were able to use an SSH-encrypted tunnel to securely transfer the data from the local system to the remote host. Second, we specified the remote host by prefixing its name (in this case the remote host is named *remote-sys*) to the destination pathname.

The second way that `rsync` can be used to synchronize files over a network is by using an *rysnc server*. `rsync` can be configured to run as a daemon and listen to incoming requests for synchronization. This is often done to allow mirroring of a remote system. For example, Red Hat Software maintains a large repository of software packages under development for its Fedora distribution. It is useful for software testers to mirror this collection during the testing phase of the distribution release cycle. Since files in the repository change frequently (often more than once a day), it is desirable to maintain a local mirror by periodic synchronization, rather than by bulk copying of the repository. One of these repositories is kept at Georgia Tech; we could mirror it using our local copy of `rsync` and Georgia Tech's rsync server like this:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av -delete rsync://rsync.gtlib.gatech.edu/fedora-
linux-core/development/i386/os fedora-devel
```

In this example, we use the URI of the remote rsync server, which consists of a protocol (*rsync://*), followed by the remote hostname (*rsync.gtlib .gatech.edu*), followed by the pathname of the repository.

19
REGULAR EXPRESSIONS

In the next few chapters, we are going to look at tools used to manipulate text. As we have seen, text data plays an important role on all Unix-like systems, such as Linux. But before we can fully appreciate all of the features offered by these tools, we have to examine a technology that is frequently associated with the most sophisticated uses of these tools—regular expressions.

As we have navigated the many features and facilities offered by the command line, we have encountered some truly arcane shell features and commands, such as shell expansion and quoting, keyboard shortcuts, and command history, not to mention the `vi` editor. Regular expressions continue this "tradition" and may be (arguably) the most arcane feature of them all. This is not to suggest that the time it takes to learn about them is not worth the effort. Quite the contrary. A good understanding will enable us to perform amazing feats, though their full value may not be immediately apparent.

What Are Regular Expressions?

Simply put, *regular expressions* are symbolic notations used to identify patterns in text. In some ways, they resemble the shell's wildcard method of matching file- and pathnames but on a much grander scale. Regular expressions are supported by many command-line tools and by most programming languages to facilitate the solution of text manipulation problems. However, to further confuse things, not all regular expressions are the same; they vary slightly from tool to tool and from programming language to language. For our discussion, we will limit ourselves to regular expressions as described in the POSIX standard (which will cover most of the command-line tools), as opposed to many programming languages (most notably Perl), which use slightly larger and richer sets of notations.

grep—Search Through Text

The main program we will use to work with regular expressions is our old pal, `grep`. The name *grep* is actually derived from the phrase *global regular expression print*, so we can see that `grep` has something to do with regular expressions. In essence, `grep` searches text files for the occurrence of a specified regular expression and outputs any line containing a match to standard output.

So far, we have used `grep` with fixed strings, like so:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

This will list all the files in the */usr/bin* directory whose names contain the substring `zip`.

The `grep` program accepts options and arguments this way:

```
grep [options] regex [file...]
```

where *`regex`* is a regular expression.

Table 19-1 lists the commonly used `grep` options.

Table19-1: grep Options

Option	Description
`-i`	Ignore case. Do not distinguish between upper- and lowercase characters. May also be specified `--ignore-case`.
`-v`	Invert match. Normally, `grep` prints lines that contain a match. This option causes `grep` to print every line that does not contain a match. May also be specified `--invert-match`.
`-c`	Print the number of matches (or non-matches if the `-v` option is also specified) instead of the lines themselves. May also be specified `--count`.

Table 19-1 (*continued*)

Option	Description
-l	Print the name of each file that contains a match instead of the lines themselves. May also be specified --files-with-matches.
-L	Like the -l option, but print only the names of files that do not contain matches. May also be specified --files-without-match.
-n	Prefix each matching line with the number of the line within the file. May also be specified --line-number.
-h	For multifile searches, suppress the output of filenames. May also be specified --no-filename.

In order to more fully explore grep, let's create some text files to search:

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt   dirlist-sbin.txt     dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

We can perform a simple search of our list of files like this:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

In this example, grep searches all of the listed files for the string bzip and finds two matches, both in the file *dirlist-bin.txt*. If we were interested in only the files that contained matches rather than the matches themselves, we could specify the -l option:

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Conversely, if we wanted to see a list of only the files that did not contain a match, we could do this:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Metacharacters and Literals

While it may not seem apparent, our grep searches have been using regular expressions all along, albeit very simple ones. The regular expression bzip is taken to mean that a match will occur only if the line in the file contains at least four characters and that somewhere in the line the characters *b*, *z*, *i*, and *p* are found in that order, with no other characters in between. The characters in the string `bzip` are all *literal characters*, in that they match themselves. In addition to literals, regular expressions may also include *metacharacters*, which are used to specify more complex matches. Regular expression metacharacters consist of the following:

```
^ $ . [ ] { } - ? * + ( ) | \
```

All other characters are considered literals, though the backslash character is used in a few cases to create *metasequences*, as well as allowing the metacharacters to be escaped and treated as literals instead of being interpreted as metacharacters.

Note: *As we can see, many of the regular-expression metacharacters are also characters that have meaning to the shell when expansion is performed. When we pass regular expressions containing metacharacters on the command line, it is vital that they be enclosed in quotes to prevent the shell from attempting to expand them.*

The Any Character

The first metacharacter we will look at is the dot or period character, which is used to match any character. If we include it in a regular expression, it will match any character in that character position. Here's an example:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

We searched for any line in our files that matches the regular expression `.zip`. There are a couple of interesting things to note about the results. Notice that the `zip` program was not found. This is because the inclusion of the dot metacharacter in our regular expression increased the length of the required match to four characters; because the name *zip* contains only three, it does not match. Also, if any files in our lists had contained the file extension *.zip*, they would have been matched, because the period character in the file extension is treated as "any character," too.

Anchors

The caret (^) and dollar sign ($) characters are treated as *anchors* in regular expressions. This means that they cause the match to occur only if the regular expression is found at the beginning of the line (^) or at the end of the line ($).

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Here we searched the list of files for the string zip located at the beginning of the line, the end of the line, and on a line where it is at both the beginning and the end of the line (i.e., by itself on the line.) Note that the regular expression ^$ (a beginning and an end with nothing in between) will match blank lines.

A CROSSWORD PUZZLE HELPER

My wife loves crossword puzzles, and she will sometimes ask me for help with a particular question. Something like, "What's a five-letter word whose third letter is *j* and last letter is *r* that means . . . ?" This kind of question got me thinking.

Did you know that your Linux system contains a dictionary? It does. Take a look in the */usr/share/dict* directory and you might find one, or several. The dictionary files located there are just long lists of words, one per line, arranged in alphabetical order. On my system, the *words* file contains just over 98,500 words. To find possible answers to the crossword puzzle question above, we could do this:

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

Using this regular expression, we can find all the words in our dictionary file that are five letters long and have a *j* in the third position and an *r* in the last position.

Bracket Expressions and Character Classes

In addition to matching any character at a given position in our regular expression, we can also match a single character from a specified set of characters by using *bracket expressions*. With bracket expressions, we can specify a set of characters (including characters that would otherwise be interpreted as metacharacters) to be matched. In this example, using a two-character set, we match any line that contains the string `bzip` or `gzip`:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
bzip2recover
gzip
```

A set may contain any number of characters, and metacharacters lose their special meaning when placed within brackets. However, there are two cases in which metacharacters are used within bracket expressions and have different meanings. The first is the caret (`^`), which is used to indicate negation; the second is the dash (`-`), which is used to indicate a character range.

Negation

If the first character in a bracket expression is a caret (`^`), the remaining characters are taken to be a set of characters that must not be present at the given character position. We do this by modifying our previous example:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

With negation activated, we get a list of files that contain the string `zip` preceded by any character except *b* or *g*. Notice that the file *zip* was not found. A negated character set still requires a character at the given position, but the character must not be a member of the negated set.

The caret character invokes negation only if it is the first character within a bracket expression; otherwise, it loses its special meaning and becomes an ordinary character in the set.

Traditional Character Ranges

If we wanted to construct a regular expression that would find every file in our lists whose name begins with an uppercase letter, we could do this:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

It's just a matter of putting all 26 uppercase letters in a bracket expression. But the idea of all that typing is deeply troubling, so there is another way:

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

By using a 3-character range, we can abbreviate the 26 letters. Any range of characters can be expressed this way, including multiple ranges such as this expression, which matches all filenames starting with letters and numbers:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

In character ranges, we see that the dash character is treated specially, so how do we actually include a dash character in a bracket expression? By making it the first character in the expression. Consider

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

This will match every filename containing an uppercase letter. This, on the other hand,

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

will match every filename containing a dash, an uppercase *A*, or an uppercase *Z*.

POSIX Character Classes

The traditional character ranges are an easily understood and effective way to handle the problem of quickly specifying sets of characters. Unfortunately, they don't always work. While we have not encountered any problems with our use of grep so far, we might run into problems using other programs.

Back in Chapter 4, we looked at how wildcards are used to perform pathname expansion. In that discussion, we said that character ranges could be used in a manner almost identical to the way they are used in regular expressions, but here's the problem:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

(Depending on the Linux distribution, we will get a different list of files, possibly an empty list. This example is from Ubuntu.) This command produces the expected result—a list of only the files whose names begin with an uppercase letter. But with this command we get an entirely different result (only a partial listing of the results is shown):

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

Why is that? It's a long story, but here's the short version.

Back when Unix was first developed, it only knew about ASCII characters, and this feature reflects that fact. In ASCII, the first 32 characters (numbers 0–31) are control codes (things like tabs, backspaces, and carriage returns). The next 32 (32–63) contain printable characters, including most punctuation characters and the numerals zero through nine. The next 32 (numbers 64–95) contain the uppercase letters and a few more punctuation symbols. The final 31 (numbers 96–127) contain the lowercase letters and yet more punctuation symbols. Based on this arrangement, systems using ASCII used a *collation order* that looked like this:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

This differs from proper dictionary order, which is like this:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

As the popularity of Unix spread beyond the United States, there grew a need to support characters not found in US English. The ASCII table was expanded to use a full 8 bits, adding character numbers 128–255, which accommodated many more languages. To support this ability, the POSIX standards introduced a concept called a *locale*, which could be adjusted to select the character set needed for a particular location. We can see the language setting of our system using this command:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

With this setting, POSIX-compliant applications will use a dictionary collation order rather than ASCII order. This explains the behavior of the commands above. A character range of `[A-Z]`, when interpreted in dictionary order, includes all of the alphabetic characters except the lowercase *a*—hence our results.

To partially work around this problem, the POSIX standard includes a number of character classes, which provide useful ranges of characters. They are described in Table 19-2.

Table 19-2: POSIX Character Classes

Character Class	Description
`[:alnum:]`	The alphanumeric characters; in ASCII, equivalent to `[A-Za-z0-9]`
`[:word:]`	The same as [:alnum:], with the addition of the underscore character (_)
`[:alpha:]`	The alphabetic characters; in ASCII, equivalent to `[A-Za-z]`
`[:blank:]`	Includes the space and tab characters
`[:cntrl:]`	The ASCII control codes; includes the ASCII characters 0 through 31 and 127
`[:digit:]`	The numerals 0 through 9
`[:graph:]`	The visible characters; in ASCII, includes characters 33 through 126
`[:lower:]`	The lowercase letters
`[:punct:]`	The punctuation characters; in ASCII, equivalent to `[-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~]`
`[:print:]`	The printable characters; all the characters in `[:graph:]` plus the space character
`[:space:]`	The whitespace characters including space, tab, carriage return, newline, vertical tab, and form feed; in ASCII, equivalent to `[ \t\r\n\v\f]`
`[:upper:]`	The uppercase characters
`[:xdigit:]`	Characters used to express hexadecimal numbers; in ASCII, equivalent to `[0-9A-Fa-f]`

Even with the character classes, there is still no convenient way to express partial ranges, such as `[A-M]`.

Using character classes, we can repeat our directory listing and see an improved result.

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Remember, however, that this is not an example of a regular expression; rather it is the shell performing pathname expansion. We show it here because POSIX character classes can be used for both.

REVERTING TO TRADITIONAL COLLATION ORDER

You can opt to have your system use the traditional (ASCII) collation order by changing the value of the `LANG` environment variable. As we saw in the previous section, the `LANG` variable contains the name of the language and character set used in your locale. This value was originally determined when you selected an installation language as your Linux was installed.

To see the locale settings, use the `locale` command:

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

To change the locale to use the traditional Unix behaviors, set the `LANG` variable to `POSIX`:

```
[me@linuxbox ~]$ export LANG=POSIX
```

Note that this change converts the system to use US English (more specifically, ASCII) for its character set, so be sure this is really what you want.

You can make this change permanent by adding this line to your *.bashrc* file:

```
export LANG=POSIX
```

POSIX Basic vs. Extended Regular Expressions

Just when we thought this couldn't get any more confusing, we discover that POSIX also splits regular expression implementations into two kinds: *basic regular expressions (BRE)* and *extended regular expressions (ERE)*. The features we have covered so far are supported by any application that is POSIX compliant and implements BRE. Our `grep` program is one such program.

What's the difference between BRE and ERE? It's a matter of metacharacters. With BRE, the following metacharacters are recognized: `^ $ . [ ] *` All other characters are considered literals. With ERE, the following metacharacters (and their associated functions) are added: `( ) { } ? + |`

However (and this is the fun part), the characters `()` `{}` are treated as metacharacters in BRE *if* they are escaped with a backslash, whereas with ERE, preceding any metacharacter with a backslash causes it to be treated as a literal.

Since the features we are going to discuss next are part of ERE, we are going to need to use a different `grep`. Traditionally, this has been performed by the `egrep` program, but the GNU version of `grep` also supports extended regular expressions when the `-E` option is used.

POSIX

During the 1980s, Unix became a very popular commercial operating system, but by 1988, the Unix world was in turmoil. Many computer manufacturers had licensed the Unix source code from its creators AT&T, and were supplying various versions of the operating system with their systems. However, in their efforts to create product differentiation, each manufacturer added proprietary changes and extensions. This started to limit the compatibility of the software. As always with proprietary vendors, each was trying to play a winning game of "lock-in" with their customers. This dark time in the history of Unix is known today as *the Balkanization*.

Enter the IEEE (Institute of Electrical and Electronics Engineers). In the mid-1980s, the IEEE began developing a set of standards that would define how Unix (and Unix-like) systems would perform. These standards, formally known as IEEE 1003, define the *application programming interfaces (APIs)*, the shell and utilities that are to be found on a standard Unix-like system. The name *POSIX*, which stands for *Portable Operating System Interface* (with the *X* added to the end for extra snappiness), was suggested by Richard Stallman (yes, *that* Richard Stallman) and was adopted by the IEEE.

Alternation

The first of the extended regular expression features we will discuss is called *alternation*, which is the facility that allows a match to occur from among a set of expressions. Just as a bracket expression allows a single character to match from a set of specified characters, alternation allows matches from a set of strings or other regular expressions.

To demonstrate, we'll use `grep` in conjunction with `echo`. First, let's try a plain old string match:

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

A pretty straightforward example, in which we pipe the output of `echo` into `grep` and see the results. When a match occurs, we see it printed out; when no match occurs, we see no results.

Now we'll add alternation, signified by the vertical pipe metacharacter:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Here we see the regular expression `'AAA|BBB'`, which means "match either the string `AAA` or the string `BBB`." Notice that since this is an extended feature, we added the `-E` option to `grep` (though we could have used the `egrep` program instead), and we enclosed the regular expression in quotes to prevent the shell from interpreting the vertical pipe metacharacter as a pipe operator. Alternation is not limited to two choices:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

To combine alternation with other regular-expression elements, we can use `()` to separate the alternation:

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

This expression will match the filenames in our lists that start with either `bz`, `gz`, or `zip`. If we leave off the parentheses, the meaning of this regular expression changes to match any filename that begins with `bz` *or contains* `gz` *or contains* `zip`:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

Quantifiers

Extended regular expressions support several ways to specify the number of times an element is matched.

?—Match an Element Zero Times or One Time

This quantifier means, in effect, "Make the preceding element optional." Let's say we wanted to check a phone number for validity and we considered a phone number to be valid if it matched either of these two forms, `(nnn) nnn-nnnn` or `nnn nnn-nnnn`, where *n* is a numeral. We could construct a regular expression like this:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

In this expression, we follow the parentheses characters with question marks to indicate that they are to be matched zero or one time. Again, since the parentheses are normally metacharacters (in ERE), we precede them with backslashes to cause them to be treated as literals instead.

Let's try it:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
[0-9][0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Here we see that the expression matches both forms of the phone number but does not match one containing non-numeric characters.

*—Match an Element Zero or More Times

Like the ? metacharacter, the * is used to denote an optional item; however, unlike the ?, the item may occur any number of times, not just once. Let's say we want to see if a string is a sentence; that is, it starts with an uppercase letter, then contains any number of upper- and lowercase letters and spaces, and ends with a period. To match this (very crude) definition of a sentence, we could use a regular expression like this:

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

The expression consists of three items: a bracket expression containing the `[:upper:]` character class, a bracket expression containing both the `[:upper:]` and `[:lower:]` character classes and a space, and a period escaped with a backslash. The second element is trailed with an * metacharacter so that after the leading uppercase letter in our sentence, any number of upper- and lowercase letters and spaces may follow it and still match:

```
[me@linuxbox ~]$ echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:]
 ]*\.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '[[:upper:]][[:upper:][:lower:]
 ]*\.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '[[:upper:]][[:upper:][:lower:
] ]*\.'
[me@linuxbox ~]$
```

The expression matches the first two tests, but not the third, since it lacks the required leading uppercase character and trailing period.

+—Match an Element One or More Times

The + metacharacter works much like the *, except it requires at least one instance of the preceding element to cause a match. Here is a regular expression that will match only lines consisting of groups of one or more alphabetic characters separated by single spaces:

```
^([[:alpha:]]+ ?)+$
```

Let's try it:

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc  d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

We see that this expression does not match the line `"a b 9"`, because it contains a non-alphabetic character; nor does it match `"abc  d"`, because more than one space character separates the characters *c* and *d*.

{ }—Match an Element a Specific Number of Times

The { and } metacharacters are used to express minimum and maximum numbers of required matches. They may be specified in four possible ways, as shown in Table 19-3.

Table 19-3: Specifying the Number of Matches

Specifier	Meaning
`{`*n*`}`	Match the preceding element if it occurs exactly *n* times.
`{`*n*`,`*m*`}`	Match the preceding element if it occurs at least *n* times, but no more than *m* times.
`{`*n*`,}`	Match the preceding element if it occurs *n* or more times.
`{,`*m*`}`	Match the preceding element if it occurs no more than *m* times.

Going back to our earlier example with the phone numbers, we can use this method of specifying repetitions to simplify our original regular expression from

```
^\(?[0-9][0-9][0-9]\)?  [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

to

```
^\(?[0-9]{3}\)?  [0-9]{3}-[0-9]{4}$
```

Let's try it:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-
9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9]
{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-[0-9
]{4}$'
[me@linuxbox ~]$
```

As we can see, our revised expression can successfully validate numbers both with and without the parentheses, while rejecting those numbers that are not properly formatted.

Putting Regular Expressions to Work

Let's look at some of the commands we already know and see how they can be used with regular expressions.

Validating a Phone List with grep

In our earlier example, we looked at single phone numbers and checked them for proper formatting. A more realistic scenario would be checking a list of numbers instead, so let's make a list. We'll do this by reciting a magical incantation to the command line. It will be magic because we have not covered most of the commands involved, but worry not—we will get there in future chapters. Here is the incantation:

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDOM:0:3}-$
{RANDOM:0:4}" >> phonelist.txt; done
```

This command will produce a file named *phonelist.txt* containing 10 phone numbers. Each time the command is repeated, another 10 numbers are added to the list. We can also change the value `10` near the beginning of the command to produce more or fewer phone numbers. If we examine the contents of the file, however, we see we have a problem:

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

Some of the numbers are malformed, which is perfect for our purposes because we will use `grep` to validate them.

One useful method of validation would be to scan the file for invalid numbers and display the resulting list.

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

Here we use the `-v` option to produce an inverse match so that we will output only the lines in the list that do not match the specified expression.

The expression itself includes the anchor metacharacters at each end to ensure that the number has no extra characters at either end. This expression also requires that the parentheses be present in a valid number, unlike our earlier phone number example.

Finding Ugly Filenames with find

The `find` command supports a test based on a regular expression. There is an important consideration to keep in mind when using regular expressions in `find` versus `grep`. Whereas `grep` will print a line when the line *contains* a string that matches an expression, `find` requires that the pathname *exactly match* the regular expression. In the following example, we will use `find` with a regular expression to find every pathname that contains any character that is not a member of the following set:

```
[-_./0-9a-zA-Z]
```

Such a scan would reveal pathnames that contain embedded spaces and other potentially offensive characters:

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Due to the requirement for an exact match of the entire pathname, we use `.*` at both ends of the expression to match zero or more instances of any character. In the middle of the expression, we use a negated bracket expression containing our set of acceptable pathname characters.

Searching for Files with locate

The `locate` program supports both basic (the `--regexp` option) and extended (the `--regex` option) regular expressions. With it, we can perform many of the same operations that we performed earlier with our *dirlist* files:

```
[me@linuxbox ~]$ locate --regex 'bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Using alternation, we perform a search for pathnames that contain either *bin/bz*, *bin/gz*, or */bin/zip*.

Searching for Text with less and vim

`less` and `vim` share the same method of searching for text. Pressing the / key followed by a regular expression will perform a search. We use `less` to view our *phonelist.txt* file:

```
[me@linuxbox ~]$ less phonelist.txt
```

Then we search for our validation expression:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

`less` will highlight the strings that match, leaving the invalid ones easy to spot:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

`vim`, on the other hand, supports basic regular expressions, so our search expression would look like this:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

We can see that the expression is mostly the same; however, many of the characters that are considered metacharacters in extended expressions are considered literals in basic expressions. They are treated as metacharacters only when escaped with a backslash. Depending on the particular configuration of `vim` on our system, the matching will be highlighted. If not, try the command-mode command `:hlsearch` to activate search highlighting.

Note: *Depending on your distribution,* `vim` *may or may not support text-search highlighting. Ubuntu, in particular, supplies a very stripped-down version of* `vim` *by default. On such systems, you may want to use your package manager to install a more complete version of* `vim`*.*

Final Note

In this chapter, we've seen a few of the many uses of regular expressions. We can find even more if we use regular expressions to search for additional applications that use them. We can do that by searching the man pages:

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

The `zgrep` program provides a frontend for `grep`, allowing it to read compressed files. In our example, we search the compressed Section 1 man page files in their usual location. The result of this command is a list of files containing the string `regex` or `regular expression`. As we can see, regular expressions show up in a lot of programs.

There is one feature found in basic regular expressions that we did not cover. Called *back references*, this feature will be discussed in the next chapter.

20
TEXT PROCESSING

All Unix-like operating systems rely heavily on text files for several types of 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 spell checker.

Applications of Text

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

Documents

Many people write documents using plaintext 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 use 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 *HTML (Hypertext Markup Language)* or *XML (Extensible Markup Language)* as a markup language 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 plaintext 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—Concatenate Files and Print on Standard Output

The `cat` program has a number of interesting options. Many of them are used to 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 if 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, 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 will 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 common notation 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 a several ways to convert files from DOS to Unix format. On many Linux systems, programs called dos2unix and unix2dos 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 very simple; it simply 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—Sort Lines of Text Files

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.

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

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

Since 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 shows 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`	Makes sorting case insensitive.
`-n`	`--numeric-sort`	Performs 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.
`-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 to standard output.
`-t`	`--field-separator=`*`char`*	Define the field-separator character. By default, fields are separated by spaces or tabs.

Although most of the options above 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. 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/apmd
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 -nr 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, the result of ls -l looks like this:

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root   root     34824 2012-04-04 02:42 [
-rwxr-xr-x 1 root   root    101556 2011-11-27 06:08 a2p
-rwxr-xr-x 1 root   root     13036 2012-02-27 08:22 aconnect
-rwxr-xr-x 1 root   root     10552 2011-08-15 10:34 acpi
-rwxr-xr-x 1 root   root      3800 2012-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root   root      7536 2012-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root   root      3576 2012-04-29 07:57 addpart
-rwxr-xr-x 1 root   root     20808 2012-01-03 18:02 addr2line
-rwxr-xr-x 1 root   root    489704 2012-10-09 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 -nr -k 5 | head
-rwxr-xr-x 1 root   root    8234216 2012-04-07 17:42 inkscape
-rwxr-xr-x 1 root   root    8222692 2012-04-07 17:42 inkview
-rwxr-xr-x 1 root   root    3746508 2012-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root   root    3654020 2012-08-26 16:16 quanta
-rwxr-xr-x 1 root   root    2928760 2012-09-10 14:31 gdbtui
-rwxr-xr-x 1 root   root    2928756 2012-09-10 14:31 gdb
-rwxr-xr-x 1 root   root    2602236 2012-10-10 12:56 net
-rwxr-xr-x 1 root   root    2304684 2012-10-10 12:56 rpcclient
-rwxr-xr-x 1 root   root    2241832 2012-04-04 05:56 aptitude
-rwxr-xr-x 1 root   root    2202476 2012-10-10 12:56 smbcacls
```

Many uses of sort involve the processing of *tabular data*, such as the results of the ls command above. If we apply database terminology to the table above, 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 example above, 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 very interesting and has many features, but first we need to talk about how sort defines fields. Let's consider a very 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 the characters William and the second field contains the characters   Shotts, meaning 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, 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 2012-04-07 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, the version number, and the 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 the 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. Since a 1 comes before a 5 in the character set, version 10 ends up at the top while version 9 falls to the bottom.

To fix this problem, we have to sort on multiple keys. We want to perform an alphabetic sort on the first field and then a numeric sort on the third 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. Since 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 that 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; take, for example, 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—Report or Omit Repeated Lines

Compared to sort, the uniq program is a lightweight. uniq performs a seemingly trivial task. When given a sorted file (including 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.

Note: *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 out:

```
[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, the results are no different from our original file; the duplicates were not removed:

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

For uniq to actually 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	Description
`-c`	Output a list of duplicate lines preceded by the number of times the line occurs.
`-d`	Output only repeated lines, rather than unique lines.
`-f` *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 alternative field separator.
`-i`	Ignore case during the line comparisons.
`-s` *n*	Skip (ignore) the leading *n* characters of each line.
`-u`	Output only unique lines. This is the default.

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 shown in Table 20-3.

Table 20-3: cut Selection Options

Option	Description
`-c` *`char_list`*	Extract the portion of the line defined by *`char_list`*. The list may consist of one or more comma-separated numerical ranges.
`-f` *`field_list`*	Extract one or more fields from the line as defined by *`field_list`*. The list may contain one or more fields or field ranges separated by commas.
`-d` *`delim_char`*	When `-f` is specified, use *`delim_char`* as the field delimiting character. By default, fields must be separated by a single tab character.
`--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 if it is "clean" enough to be a good specimen for our `cut` examples. If we use `cat` with the `-A` option, we can see if 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—no embedded spaces, just single tab characters between the fields. Since 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 example above, 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 characters, 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 it outputs the modified text to standard output.

If we process our *distros.txt* file with expand, we can use the 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 of 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 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 does so in a unique way. 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 very 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, as 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 field for this demonstration) and the release names:

```
[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:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt > distros-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, displaying 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, the lines unique to the second file argument; and the third column, 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 column(s) to suppress. For example, if we wanted to output only the lines shared by both files, we would suppress the output of columns 1 and 2:

```
[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 because it 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, we see its default style of output: a terse description of the differences between the two files.

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

In the default format, each group of changes is preceded by a *change command* (see Table 20-4) 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.

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), the output looks like this:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt   2012-12-23 06:40:13.000000000 -0500
--- file2.txt   2012-12-23 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 `*** 1,4 ****`, which indicates lines 1 through 4 in the first file. Later we see `--- 1,4 ----`, which indicates lines 1 through 4 in the second file. Within a change group, lines begin with one of four indicators, as shown in Table 20-5.

Table 20-5: diff Context-Format Change Indicators

Indicator	Meaning
(none)	A line shown for context. It does not indicate a difference between the two files.
-	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    2012-12-23 06:40:13.000000000 -0500
+++ file2.txt    2012-12-23 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 example above, 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. As shown in Table 20-6, each line starts with one of three possible characters.

Table 20-6: diff Unified-Format Change Indicators

Character	Meaning
(none)	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 of 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 his own source tree. Using diff/patch offers two significant advantages:

- The diff file is very 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 additional utility programs 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 (due to the locale collation order) and thus should be used with caution.
- POSIX character classes; for example, [:upper:].

In most cases, the character sets should be of equal length; however, it is possible for the first set to be larger than the second, particularly if we wish 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

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

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. Since this is halfway up the possible 26 characters, performing the algorithm a second time on the text restores it to its original form. 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 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: *http://en.wikipedia.org/wiki/ROT13*.

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
```

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 very 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 example above, 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
```

When the underscore character is used 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. We can specify another number:

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

Now we see that the editing is not carried out, because our input stream does not have a line 2.

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

Table 20-7: sed Address Notation

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

We'll demonstrate different kinds of addresses using the *distros.txt* file from earlier in this chapter. First, 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 autoprint 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 of 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 is a more complete list of the basic editing commands.

Table 20-8: sed Basic Editing Commands

Command	Description
`=`	Output 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 edits only 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.

Table 20-8 (*continued*)

Command	Description
`Q`	Exit `sed` without processing any more lines.
`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 on 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 before 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. To perform 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. Since 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}$
```

which 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 one to nine, 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
```

which 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 since `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/' dis
tros.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 given sed single commands only 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*. As in 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 through 6 contain text to be inserted at the address 1, the first line of the input. The `i` command is followed by the sequence backslash–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` command and the commands `a` (which appends text) and `c` (which replaces text) allow multiple lines of text, providing that 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 character classes. Again, since 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 very 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 programs covered here and extend into the realm of complete programming languages. perl, in particular, is often used in place of shell scripts for many system-management and administration tasks, as well as being a very 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 very good tools for the Linux command line user.

aspell—Interactive Spell Checker

The last tool we will look at is aspell, an interactive spellchecker. The aspell program is the successor to an earlier program named ispell, and it 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 very effectively as a stand-alone tool from the command line. It has the ability to intelligently check various type of text files, including HTML documents, C/C++ programs, email messages, and other kinds of specialized texts.

To spellcheck a text file containing simple prose, aspell 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 very bottom, we see a prompt ready to accept our choice.

If we enter `1`, `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
```

Our result is 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.*

Final Note

In this chapter, we have 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 semi-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).

21 FORMATTING OUTPUT

In this chapter, we continue our look at text-related tools, focusing on programs that are used to format text output rather than change the text itself. These tools are often used to prepare text for printing, a subject that we will cover in the next chapter. The programs that we will cover in this chapter include the following:

- `nl`—Number lines.
- `fold`—Wrap each line to a specified length.
- `fmt`—A simple text formatter.
- `pr`—Format text for printing.
- `printf`—Format and print data.
- `groff`—A document formatting system.

Simple Formatting Tools

We'll look at some of the simple formatting tools first. These are mostly single-purpose programs, and a bit unsophisticated in what they do, but they can be used for small tasks and as parts of pipelines and scripts.

nl—Number Lines

The `nl` program is a rather arcane tool used to perform a simple task: It numbers lines. In its simplest use, it resembles `cat -n`:

```
[me@linuxbox ~]$ nl distros.txt | head
     1  SUSE            10.2    12/07/2006
     2  Fedora          10      11/25/2008
     3  SUSE            11.0    06/19/2008
     4  Ubuntu          8.04    04/24/2008
     5  Fedora          8       11/08/2007
     6  SUSE            10.3    10/04/2007
     7  Ubuntu          6.10    10/26/2006
     8  Fedora          7       05/31/2007
     9  Ubuntu          7.10    10/18/2007
    10  Ubuntu          7.04    04/19/2007
```

Like `cat`, `nl` can accept either multiple filenames as command-line arguments or standard input. However, `nl` has a number of options and supports a primitive form of markup to allow more complex kinds of numbering.

`nl` supports a concept called *logical pages* when numbering. This allows `nl` to reset (start over) the numerical sequence when numbering. Using options, it is possible to set the starting number to a specific value and, to a limited extent, set its format. A logical page is further broken down into a header, body, and footer. Within each of these sections, line numbering may be reset and/or be assigned a different style. If `nl` is given multiple files, it treats them as a single stream of text. Sections in the text stream are indicated by the presence of some rather odd-looking markup added to the text, as shown in Table 21-1.

Table 21-1: nl Markup

Markup	Meaning
`\:\:\:`	Start of logical-page header
`\:\:`	Start of logical-page body
`\:`	Start of logical-page footer

Each of the markup elements in Table 21-1 must appear alone on its own line. After processing a markup element, `nl` deletes it from the text stream.

Table 21-2 lists the common options for `nl`.

Table 21-2: Common nl Options

Option	Meaning
`-b` *`style`*	Set body numbering to *`style`*, where *`style`* is one of the following: • **`a`** Number all lines. • **`t`** Number only non-blank lines. This is the default. • **`n`** None. • ***`pregexp`*** Number only lines matching basic regular expression *`regexp`*.
`-f` *`style`*	Set footer numbering to *`style`*. Default is n (none).
`-h` *`style`*	Set header numbering to *`style`*. Default is n (none).
`-i` *`number`*	Set page numbering increment to *`number`*. Default is 1.
`-n` *`format`*	Set numbering format to *`format`*, where *`format`* is one of the following: • **`ln`** Left justified, without leading zeros. • **`rn`** Right justified, without leading zeros. This is the default. • **`rz`** Right justified, with leading zeros.
`-p`	Do not reset page numbering at the beginning of each logical page.
`-s` *`string`*	Add *`string`* to the end of each line number to create a separator. Default is a single tab character.
`-v` *`number`*	Set first line number of each logical page to *`number`*. Default is 1.
`-w` *`width`*	Set width of the line number field to *`width`*. Default is 6.

Admittedly, we probably won't be numbering lines that often, but we can use `nl` to look at how we can combine multiple tools to perform more complex tasks. We will build on our work in the previous chapter to produce a Linux distributions report. Since we will be using `nl`, it will be useful to include its header/body/footer markup. To do this, we will add it to the `sed` script from the last chapter. Using our text editor, we will change the script as follows and save it as *distros-nl.sed*:

```
# sed script to produce Linux distributions report

1 i\
\\:\\:\\:\
\
Linux Distributions Report\
\
Name                Ver.    Released\
----                ----    --------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
```

```
$ a\
\\:\
\
End Of Report
```

The script now inserts the `nl` logical-page markup and adds a footer at the end of the report. Note that we had to double up the backslashes in our markup, because `sed` normally interprets them as escape characters.

Next, we'll produce our enhanced report by combining `sort`, `sed`, and `nl`:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.sed | nl


       Linux Distributions Report

        Name    Ver.    Released
        ----    ----    --------

     1  Fedora  5       2006-03-20
     2  Fedora  6       2006-10-24
     3  Fedora  7       2007-05-31
     4  Fedora  8       2007-11-08
     5  Fedora  9       2008-05-13
     6  Fedora  10      2008-11-25
     7  SUSE    10.1    2006-05-11
     8  SUSE    10.2    2006-12-07
     9  SUSE    10.3    2007-10-04
    10  SUSE    11.0    2008-06-19
    11  Ubuntu  6.06    2006-06-01
    12  Ubuntu  6.10    2006-10-26
    13  Ubuntu  7.04    2007-04-19
    14  Ubuntu  7.10    2007-10-18
    15  Ubuntu  8.04    2008-04-24
    16  Ubuntu  8.10    2008-10-30



       End Of Report
```

Our report is the result of our pipeline of commands. First, we sort the list by distribution name and version (fields 1 and 2), and then we process the results with `sed`, adding the report header (including the logical page markup for `nl`) and footer. Finally, we process the result with `nl`, which, by default, numbers only the lines of the text stream that belong to the body section of the logical page.

We can repeat the command and experiment with different options for `nl`. Some interesting ones are

```
nl -n rz
```

and

```
nl -w 3 -s ' '
```

fold—Wrap Each Line to a Specified Length

Folding is the process of breaking lines of text at a specified width. Like our other commands, fold accepts either one or more text files or standard input. If we send fold a simple stream of text, we can see how it works:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." | fold
-w 12
The quick br
own fox jump
ed over the
lazy dog.
```

Here we see fold in action. The text sent by the echo command is broken into segments specified by the -w option. In this example, we specify a line width of 12 characters. If no width is specified, the default is 80 characters. Notice that the lines are broken regardless of word boundaries. The addition of the -s option will cause fold to break the line at the last available space before the line width is reached:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." | fold
-w 12 -s
The quick
brown fox
jumped over
the lazy
dog.
```

fmt—A Simple Text Formatter

The fmt program also folds text, plus a lot more. It accepts either files or standard input and performs paragraph formatting on the text stream. Basically, it fills and joins lines in text while preserving blank lines and indentation.

To demonstrate, we'll need some text. Let's lift some from the fmt info page:

```
   `fmt' reads from the specified FILE arguments (or standard input if none
are given), and writes to standard output.

   By default, blank lines, spaces between words, and indentation are
preserved in the output; successive input lines with different
indentation are not joined; tabs are expanded on input and introduced on
output.

   `fmt' prefers breaking lines at the end of a sentence, and tries to avoid
line breaks after the first word of a sentence or before the last word of a
sentence. A "sentence break" is defined as either the end of a paragraph or a
word ending in any of `.?!', followed by two spaces or end of line, ignoring
any intervening parentheses or quotes. Like TeX, `fmt' reads entire
"paragraphs" before choosing line breaks; the algorithm is a variant of that
given by Donald E. Knuth and Michael F. Plass in "Breaking Paragraphs Into
Lines", `Software--Practice & Experience' 11, 11 (November 1981), 1119-1184.
```

We'll copy this text into our text editor and save the file as *fmt-info.txt*. Now, let's say we wanted to reformat this text to fit a 50-character-wide column. We could do this by processing the file with `fmt` and the `-w` option:

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
   `fmt' reads from the specified FILE arguments
   (or standard input if
none are given), and writes to standard output.

   By default, blank lines, spaces between words,
   and indentation are
preserved in the output; successive input lines
with different indentation are not joined; tabs
are expanded on input and introduced on output.
```

Well, that's an awkward result. Perhaps we should actually read this text, since it explains what's going on:

> By default, blank lines, spaces between words, and indentation are preserved in the output; successive input lines with different indentation are not joined; tabs are expanded on input and introduced on output.

So, `fmt` is preserving the indentation of the first line. Fortunately, `fmt` provides an option to correct this:

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
   `fmt' reads from the specified FILE arguments
(or standard input if none are given), and writes
to standard output.

   By default, blank lines, spaces between words,
and indentation are preserved in the output;
successive input lines with different indentation
are not joined; tabs are expanded on input and
introduced on output.

   `fmt' prefers breaking lines at the end of a
sentence, and tries to avoid line breaks after
the first word of a sentence or before the
last word of a sentence. A "sentence break"
is defined as either the end of a paragraph
or a word ending in any of `.?!', followed


by two spaces or end of line, ignoring any
intervening parentheses or quotes. Like TeX,
`fmt' reads entire "paragraphs" before choosing
line breaks; the algorithm is a variant of
that given by Donald E. Knuth and Michael F.
Plass in "Breaking Paragraphs Into Lines",
`Software--Practice & Experience' 11, 11
(November 1981), 1119-1184.
```

Much better. By adding the `-c` option, we now have the desired result.

`fmt` has some interesting options, as shown in Table 21-3.

Table 21-3: fmt Options

Option	Description
`-c`	Operate in *crown margin* mode. This preserves the indentation of the first two lines of a paragraph. Subsequent lines are aligned with the indentation of the second line.
`-p` *`string`*	Format only those lines beginning with the prefix *`string`*. After formatting, the contents of *`string`* are prefixed to each reformatted line. This option can be used to format text in source code comments. For example, any programming language or configuration file that uses a `#` character to delineate a comment could be formatted by specifying `-p '# '` so that only the comments will be formatted. See the example below.
`-s`	Split-only mode. In this mode, lines will be split only to fit the specified column width. Short lines will not be joined to fill lines. This mode is useful when formatting text, such as code, where joining is not desired.
`-u`	Perform uniform spacing. This will apply traditional "typewriter-style" formatting to the text. This means a single space between words and two spaces between sentences. This mode is useful for removing *justification*, that is, forced alignment to both the left and right margins.
`-w` *`width`*	Format text to fit within a column *`width`* characters wide. The default is 75 characters. Note: `fmt` actually formats lines slightly shorter than the specified width to allow for line balancing.

The `-p` option is particularly interesting. With it, we can format selected portions of a file, provided that the lines to be formatted all begin with the same sequence of characters. Many programming languages use the hash mark (`#`) to indicate the beginning of a comment and thus can be formatted using this option. Let's create a file that simulates a program that uses comments:

```
[me@linuxbox ~]$ cat > fmt-code.txt
# This file contains code with comments.

# This line is a comment.
# Followed by another comment line.
# And another.

This, on the other hand, is a line of code.
And another line of code.
And another.
```

Our sample file contains comments, which begin with the string `# ` (a `#` followed by a space), and lines of "code," which do not. Now, using `fmt`, we can format the comments and leave the code untouched:

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt
# This file contains code with comments.

# This line is a comment. Followed by another
# comment line. And another.

This, on the other hand, is a line of code.
And another line of code.
And another.
```

Notice that the adjoining comment lines are joined, while the blank lines and the lines that do not begin with the specified prefix are preserved.

pr—Format Text for Printing

The `pr` program is used to *paginate* text. When printing text, it is often desirable to separate the pages of output with several lines of whitespace to provide a top and bottom margin for each page. Further, this whitespace can be used to insert a header and footer on each page.

We'll demonstrate `pr` by formatting our *distros.txt* file into a series of very short pages (only the first two pages are shown):

```
[me@linuxbox ~]$  pr -l 15 -w 65 distros.txt



2012-12-11 18:27                distros.txt                Page 1



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




2012-12-11 18:27                distros.txt                Page 2



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
```

In this example, we employ the -l option (for page length) and the -w option (page width) to define a "page" that is 65 characters wide and 15 lines long. pr paginates the contents of the *distros.txt* file, separates each page with several lines of whitespace, and creates a default header containing the file modification time, filename, and page number. The pr program provides many options to control page layout. We'll take a look at more of them in Chapter 22.

printf—Format and Print Data

Unlike the other commands in this chapter, the printf command is not used for pipelines (it does not accept standard input), nor does it find frequent application directly on the command line (it's used mostly in scripts). So why is it important? Because it is so widely used.

printf (from the phrase *print formatted*) was originally developed for the C programming language and has been implemented in many programming languages, including the shell. In fact, in bash, printf is a built-in.

printf works like this:

```
printf "format" arguments
```

The command is given a string containing a format description, which is then applied to a list of arguments. The formatted result is sent to standard output. Here is a trivial example:

```
[me@linuxbox ~]$  printf "I formatted the string: %s\n" foo
I formatted the string: foo
```

The format string may contain literal text (like I formatted the string:); escape sequences (such as \n, a newline character); and sequences beginning with the % character, which are called *conversion specifications*. In the example above, the conversion specification %s is used to format the string foo and place it in the command's output. Here it is again:

```
[me@linuxbox ~]$ printf "I formatted '%s' as a string.\n" foo
I formatted 'foo' as a string.
```

As we can see, the %s conversion specification is replaced by the string foo in the command's output. The s conversion is used to format string data. There are other specifiers for other kinds of data. Table 21-4 lists the commonly used data types.

Table 21-4: Common printf Data-Type Specifiers

Specifier	Description
d	Format a number as a signed decimal integer.
f	Format and output a floating point number.

(*continued*)

Table 21-4 (*continued*)

Specifier	Description
o	Format an integer as an octal number.
s	Format a string.
x	Format an integer as a hexadecimal number using lowercase *a–f* where needed.
X	Same as x, but use uppercase letters.
%	Print a literal % symbol (i.e., specify "%%").

We'll demonstrate the effect each of the conversion specifiers on the string 380:

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380 380 380
380, 380.000000, 574, 380, 17c, 17C
```

Since we specified six conversion specifiers, we must also supply six arguments for printf to process. The six results show the effect of each specifier.

Several optional components may be added to the conversion specifier to adjust its output. A complete conversion specification may consist of the following:

%[*flags*][*width*][.*precision*]*conversion_specification*

Multiple optional components, when used, must appear in the order specified above to be properly interpreted. Table 21-5 describes each component.

Table 21-5: printf Conversion-Specification Components

Component	Description
flags	There are five different flags: • **#** Use the *alternate format* for output. This varies by data type. For o (octal number) conversion, the output is prefixed with 0 (zero). For x and X (hexadecimal number) conversions, the output is prefixed with 0x or 0X respectively. • **0 (zero)** Pad the output with zeros. This means that the field will be filled with leading zeros, as in 000380. • **- (dash)** Left-align the output. By default, printf right-aligns output. • **(space)** Produce a leading space for positive numbers. • **+ (plus sign)** Sign positive numbers. By default, printf signs only negative numbers.

Table 21-5 (*continued*)

Component	Description
width	A number specifying the minimum field width
.precision	For floating-point numbers, specify the number of digits of precision to be output after the decimal point. For string conversion, *precision* specifies the number of characters to output.

Table 21-6 lists some examples of different formats in action.

Table 21-6: print Conversion Specification Examples

Argument	Format	Result	Notes
`380`	`"%d"`	`380`	Simple formatting of an integer
`380`	`"%#x"`	`0x17c`	Integer formatted as a hexadecimal number using the alternate format flag
`380`	`"%05d"`	`00380`	Integer formatted with leading zeros (padding) and a minimum field width of five characters
`380`	`"%05.5f"`	`380.00000`	Number formatted as a floating-point number with padding and 5 decimal places of precision. Since the specified minimum field width (5) is less than the actual width of the formatted number, the padding has no effect.
`380`	`"%010.5f"`	`0380.00000`	Increasing the minimum field width to 10 makes the padding visible.
`380`	`"%+d"`	`+380`	The + flag signs a positive number.
`380`	`"%-d"`	`380`	The - flag left-aligns the formatting.
`abcdefghijk`	`"%5s"`	`abcedfghijk`	A string is formatted with a minimum field width.
`abcdefghijk`	`"%.5s"`	`abcde`	By applying precision to a string, it is truncated.

Again, printf is used mostly in scripts, where it is employed to format tabular data, rather than on the command line directly. But we can still show how it can be used to solve various formatting problems. First, let's output some fields separated by tab characters:

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1    str2    str3
```

By inserting \t (the escape sequence for a tab), we achieve the desired effect. Next, some numbers with neat formatting:

```
[me@linuxbox ~]$ printf "Line: %05d %15.3f Result: %+15d\n" 1071 3.14156295
32589
Line: 01071           3.142 Result:          +32589
```

This shows the effect of minimum field width on the spacing of the fields. Or how about formatting a tiny web page?

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n\t</head>
\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Page Title" "Page Content"
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <p>Page Content</p>
        </body>
</html>
```

Document Formatting Systems

So far, we have examined the simple text-formatting tools. These are good for small, simple tasks, but what about larger jobs? One of the reasons that Unix became a popular operating system among technical and scientific users (aside from providing a powerful multitasking, multiuser environment for all kinds of software development) is that it offered tools that could be used to produce many types of documents, particularly scientific and academic publications. In fact, as the GNU documentation describes, document preparation was instrumental to the development of Unix:

> The first version of UNIX was developed on a PDP-7 which was sitting around Bell Labs. In 1971 the developers wanted to get a PDP-11 for further work on the operating system. In order to justify the cost for this system, they proposed that they would implement a document formatting system for the AT&T patents division. This first formatting program was a reimplementation of McIllroy's *roff*, written by J.F. Ossanna.

The roff Family and TEX

Two main families of document formatters dominate the field: those descended from the original `roff` program, including `nroff` and `troff`, and those based on Donald Knuth's TEX (pronounced "tek") typesetting system. And yes, the dropped "E" in the middle is part of its name.

The name *roff* is derived from the term *run off* as in, "I'll run off a copy for you." The `nroff` program is used to format documents for output to devices that use monospaced fonts, such as character terminals and typewriter-style printers. At the time of its introduction, this included nearly all printing devices attached to computers. The later `troff` program formats documents for output on *typesetters*, devices used to produce "camera-ready" type for commercial printing. Most computer printers today are able to simulate the output of typesetters. The `roff` family also includes some other programs that are used to prepare portions of documents. These include `eqn` (for mathematical equations) and `tbl` (for tables).

The TEX system (in stable form) first appeared in 1989 and has, to some degree, displaced `troff` as the tool of choice for typesetter output. We won't be covering TEX here, due both to its complexity (there are entire books about it) and to the fact that it is not installed by default on most modern Linux systems.

Note: *For those interested in installing TEX, check out the* `texlive` *package, which can be found in most distribution repositories, and the* `LyX` *graphical content editor.*

groff—A Document Formatting System

`groff` is a suite of programs containing the GNU implementation of `troff`. It also includes a script that is used to emulate `nroff` and the rest of the `roff` family as well.

While `roff` and its descendants are used to make formatted documents, they do it in a way that is rather foreign to modern users. Most documents today are produced using word processors that are able to perform both the composition and layout of a document in a single step. Prior to the advent of the graphical word processor, documents were often produced in a two-step process involving the use of a text editor to perform composition and a processor, such as `troff`, to apply the formatting. Instructions for the formatting program were embedded in the composed text through the use of a markup language. The modern analog for such a process is the web page, which is composed using a text editor of some kind and then rendered by a web browser using HTML as the markup language to describe the final page layout.

We're not going to cover `groff` in its entirety, as many elements of its markup language deal with rather arcane details of typography. Instead we will concentrate on one of its *macro packages* that remains in wide use. These macro packages condense many of its low-level commands into a smaller set of high-level commands that make using `groff` much easier.

For a moment, let's consider the humble man page. It lives in the */usr/share/man* directory as a gzip-compressed text file. If we were to examine its uncompressed contents, we would see the following (the man page for ls in section 1 is shown):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head
.\" DO NOT MODIFY THIS FILE!  It was generated by help2man 1.35.
.TH LS "1" "April 2008" "GNU coreutils 6.10" "User Commands"
.SH NAME
ls \- list directory contents
.SH SYNOPSIS
.B ls
[\fIOPTION\fR]... [\fIFILE\fR]...
.SH DESCRIPTION
.\" Add any additional description here
.PP
```

Compared to the man page in its normal presentation, we can begin to see a correlation between the markup language and its results:

```
[me@linuxbox ~]$ man ls | head
LS(1)                     User Commands                      LS(1)


NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

This is of interest because man pages are rendered by groff, using the mandoc macro package. In fact, we can simulate the man command with this pipeline.

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc -T ascii |
head
LS(1)                     User Commands                      LS(1)



NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

Here we use the groff program with the options set to specify the mandoc macro package and the output driver for ASCII. groff can produce output in several formats. If no format is specified, PostScript is output by default:

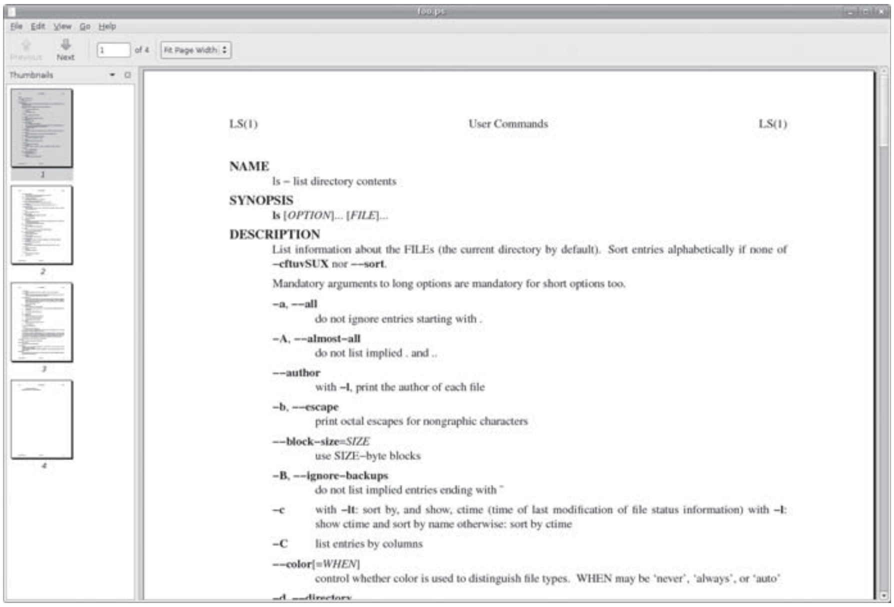
```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc | head
%!PS-Adobe-3.0
%%Creator: groff version 1.18.1
%%CreationDate: Thu Feb  2 13:44:37 2012
%%DocumentNeededResources: font Times-Roman
```

```
%%+ font Times-Bold
%%+ font Times-Italic
%%DocumentSuppliedResources: procset grops 1.18 1
%%Pages: 4
%%PageOrder: Ascend
%%Orientation: Portrait
```

PostScript is a page-description language that is used to describe the contents of a printed page to a typesetter-like device. We can take the output of our command and store it to a file (assuming that we are using a graphical desktop with a *Desktop* directory):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc > ~/Desktop
/foo.ps
```

An icon for the output file should appear on the desktop. By double-clicking the icon, a page viewer should start up and reveal the file in its rendered form (Figure 21-1).



LS(1) User Commands LS(1)

NAME
ls – list directory contents

SYNOPSIS
ls [*OPTION*]... [*FILE*]...

DESCRIPTION
List information about the FILEs (the current directory by default). Sort entries alphabetically if none of **–cftuvSUX** nor **––sort**.

Mandatory arguments to long options are mandatory for short options too.

–a, ––all
do not ignore entries starting with .

–A, ––almost–all
do not list implied . and ..

––author
with **–l**, print the author of each file

–b, ––escape
print octal escapes for nongraphic characters

––block–size=*SIZE*
use SIZE–byte blocks

–B, ––ignore–backups
do not list implied entries ending with ~

–c with **–lt**: sort by, and show, ctime (time of last modification of file status information) with **–l**: show ctime and sort by name otherwise: sort by ctime

–C list entries by columns

––color[=*WHEN*]
control whether color is used to distinguish file types. WHEN may be 'never', 'always', or 'auto'

Figure 21-1: Viewing PostScript output with a page viewer in GNOME

What we see is a nicely typeset man page for `ls`! In fact, it's possible to convert the PostScript file into a *PDF (Portable Document Format)* file with this command:

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

The `ps2pdf` program is part of the `ghostscript` package, which is installed on most Linux systems that support printing.

Note: *Linux systems often include many command line-programs for file-format conversion. They are often named using the convention* format2format. *Try using the command* `ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*` *to identify them. Also try searching for programs named* formattoformat.

For our last exercise with `groff`, we will revisit our old friend *distros.txt*. This time, we will use the `tbl` program, which is used to format tables, to typeset our list of Linux distributions. To do this, we are going to use our earlier `sed` script to add markup to a text stream that we will feed to `groff`.

First, we need to modify our `sed` script to add the necessary requests that `tbl` requires. Using a text editor, we will change *distros.sed* to the following:

```
# sed script to produce Linux distributions report

1 i\
.TS\
center box;\
cb s s\
cb cb cb\
l n c.\
Linux Distributions Report\
=\
Name	Version	Released\
_
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
.TE
```

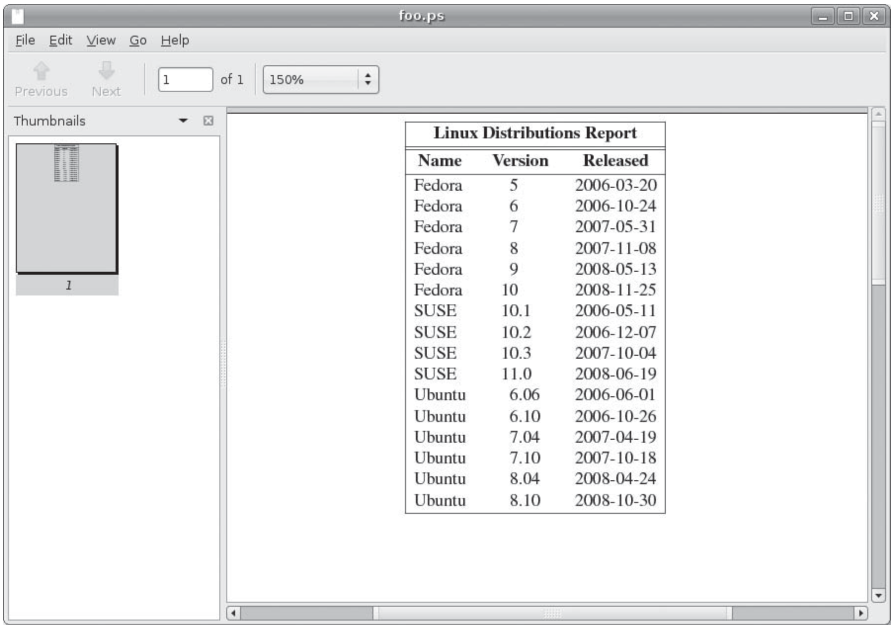
Note that for the script to work properly, care must been taken to see that the words *Name Version Released* are separated by tabs, not spaces. We'll save the resulting file as *distros-tbl.sed*. `tbl` uses the `.TS` and `.TE` requests to start and end the table. The rows following the `.TS` request define global properties of the table, which, for our example, are centered horizontally on the page and surrounded by a box. The remaining lines of the definition describe the layout of each table row. Now, if we run our report-generating pipeline again with the new `sed` script, we'll get the following :

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t -T ascii 2>/dev/null
                 +------------------------------+
                 | Linux Distributions Report   |
                 +------------------------------+
                 | Name    Version    Released  |
                 +------------------------------+
                 |Fedora     5       2006-03-20 |
                 |Fedora     6       2006-10-24 |
                 |Fedora     7       2007-05-31 |
                 |Fedora     8       2007-11-08 |
                 |Fedora     9       2008-05-13 |
                 |Fedora    10       2008-11-25 |
                 |SUSE      10.1     2006-05-11 |
                 |SUSE      10.2     2006-12-07 |
                 |SUSE      10.3     2007-10-04 |
                 |SUSE      11.0     2008-06-19 |
                 |Ubuntu     6.06    2006-06-01 |
|Ubuntu     6.10     2006-10-26 |
|Ubuntu     7.04     2007-04-19 |
|Ubuntu     7.10     2007-10-18 |
|Ubuntu     8.04     2008-04-24 |
|Ubuntu     8.10     2008-10-30 |
+-------------------------------+
```

Adding the -t option to groff instructs it to preprocess the text stream with tbl. Likewise, the -T option is used to output to ASCII rather than to the default output medium, PostScript.

The format of the output is the best we can expect if we are limited to the capabilities of a terminal screen or typewriter-style printer. If we specify PostScript output and graphically view the resulting output, we get a much more satisfying result (see Figure 21-2).

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff
-t > ~/Desktop/foo.ps
```



Linux Distributions Report		
Name	**Version**	**Released**
Fedora	5	2006-03-20
Fedora	6	2006-10-24
Fedora	7	2007-05-31
Fedora	8	2007-11-08
Fedora	9	2008-05-13
Fedora	10	2008-11-25
SUSE	10.1	2006-05-11
SUSE	10.2	2006-12-07
SUSE	10.3	2007-10-04
SUSE	11.0	2008-06-19
Ubuntu	6.06	2006-06-01
Ubuntu	6.10	2006-10-26
Ubuntu	7.04	2007-04-19
Ubuntu	7.10	2007-10-18
Ubuntu	8.04	2008-04-24
Ubuntu	8.10	2008-10-30

Figure 21-2: Viewing the finished table

Final Note

Given that text is so central to the character of Unix-like operating systems, it makes sense that there would be many tools that are used to manipulate and format text. As we have seen, there are! The simple formatting tools like fmt and pr will find many uses in scripts that produce short documents, while groff (and friends) can be used to write books. We may never write a technical paper using command-line tools (though many people do!), but it's good to know that we could.

22
PRINTING

After spending the last couple of chapters manipulating text, it's time to put that text on paper. In this chapter, we'll look at the command-line tools that are used to print files and control printer operation. We won't be looking at how to configure printing, as that varies from distribution to distribution and is usually set up automatically during installation. Note that we will need a working printer configuration to perform the exercises in this chapter.

We will discuss the following commands:

- `pr`—Convert text files for printing.
- `lpr`—Print files.
- `lp`—Print files (System V).
- `a2ps`—Format files for printing on a PostScript printer.
- `lpstat`—Show printer status information.
- `lpq`—Show printer queue status.
- `lprm`—Cancel print jobs.
- `cancel`—Cancel print jobs (System V).

A Brief History of Printing

To fully understand the printing features found in Unix-like operating systems, we must first learn some history. Printing on Unix-like systems goes way back to the beginning of the operating system itself. In those days, printers and how they were used were much different from how they are today.

Printing in the Dim Times

Like the computers themselves, printers in the pre-PC era tended to be large, expensive, and centralized. The typical computer user of 1980 worked at a terminal connected to a computer some distance away. The printer was located near the computer and was under the watchful eyes of the computer's operators.

When printers were expensive and centralized, as they often were in the early days of Unix, it was common practice for many users to share a printer. To identify print jobs belonging to a particular user, a *banner page* displaying the name of the user was often printed at the beginning of each print job. The computer support staff would then load up a cart containing the day's print jobs and deliver them to the individual users.

Character-Based Printers

The printer technology of the '80s was very different in two respects. First, printers of that period were almost always impact printers. *Impact printers* use a mechanical mechanism that strikes a ribbon against the paper to form character impressions on the page. Two of the popular technologies of that time were daisy-wheel printing and dot-matrix printing.

The second, and more important, characteristic of early printers was that they used a fixed set of characters that were intrinsic to the device itself. For example, a daisy-wheel printer could print only the characters actually molded into the petals of the daisy wheel. This made the printers much like high-speed typewriters. As with most typewriters, they printed using monospaced (fixed-width) fonts. This means that each character has the same width. Printing was done at fixed positions on the page, and the printable area of a page contained a fixed number of characters. Most printers printed 10 characters per inch (CPI) horizontally and 6 lines per inch (LPI) vertically. Using this scheme, a US-letter sheet of paper is 85 characters wide and 66 lines high. Taking into account a small margin on each side, 80 characters was considered the maximum width of a print line. This explains why terminal displays (and our terminal emulators) are normally 80 characters wide. It provides a *WYSIWYG (What You See Is What You Get)* view of printed output, using a monospaced font.

Data is sent to a typewriter-like printer in a simple stream of bytes containing the characters to be printed. For example, to print an *a*, the ASCII character code 97 is sent. In addition, the low-numbered ASCII control codes provided a means of moving the printer's carriage and paper, using codes for carriage return, line feed, form feed, and so on. Using the control codes, it's possible to achieve some limited font effects, such as boldface, by having the printer print a character, backspace, and print the character again to get a darker print impression on the page. We can actually witness this if we use `nroff` to render a man page and examine the output using `cat -A`:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat -A | head
LS(1)                     User Commands                     LS(1)
$
$
$
N^HNA^HAM^HME^HE$
      ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
      l^Hls^Hs [_^HO_^HP_^HT_^HI_^HO_^HN]... [_^HF_^HI_^HL_^HE]...$
```

The `^H` (CTRL-H) characters are the backspaces used to create the boldface effect. Likewise, we can also see a backspace/underscore sequence used to produce underlining.

Graphical Printers

The development of GUIs led to major changes in printer technology. As computers moved to more picture-based displays, printing moved from character-based to graphical techniques. This was facilitated by the advent of the low-cost laser printer, which, instead of printing fixed characters, could print tiny dots anywhere in the printable area of the page. This made printing proportional fonts (like those used by typesetters), and even photographs and high-quality diagrams, possible.

However, moving from a character-based scheme to a graphical scheme presented a formidable technical challenge. Here's why: The number of bytes needed to fill a page using a character-based printer can be calculated this way (assuming 60 lines per page, each containing 80 characters): 60 × 80 = 4,800 bytes.

In comparison, a 300-dot-per-inch (DPI) laser printer (assuming an 8-by-10-inch print area per page) requires (8 × 300) × (10 × 300) ÷ 8 = 900,000 bytes.

Many of the slow PC networks simply could not handle the nearly 1 megabyte of data required to print a full page on a laser printer, so it was clear that a clever invention was needed.

That invention turned out to be the page-description language. A *page-description language (PDL)* is a programming language that describes the contents of a page. Basically it says, "Go to this position, draw the character *a* in 10-point Helvetica, go to this position. . . ." until everything on the page is described. The first major PDL was PostScript from Adobe Systems, which is still in wide use today. The PostScript language is a complete programming language tailored for typography and other kinds of graphics and imaging. It includes built-in support for 35 standard, high-quality fonts, plus the ability to accept additional font definitions at runtime. At first, support for PostScript was built into the printers themselves. This solved the data transmission problem. While the typical PostScript program was verbose in comparison to the simple byte stream of character-based printers, it was much smaller than the number of bytes required to represent the entire printed page.

A *PostScript printer* accepted a PostScript program as input. The printer contained its own processor and memory (oftentimes making the printer a more powerful computer than the computer to which it was attached) and executed a special program called a *PostScript interpreter*, which read the incoming PostScript program and rendered the results into the printer's internal memory, thus forming the pattern of bits (dots) that would be transferred to the paper. The generic name for this process of rendering something into a large bit pattern (called a *bitmap*) is *raster image processor*, or *RIP*.

As the years went by, both computers and networks became much faster. This allowed the RIP to move from the printer to the host computer, which, in turn, permitted high-quality printers to be much less expensive.

Many printers today still accept character-based streams, but many low-cost printers do not. They rely on the host computer's RIP to provide a stream of bits to print as dots. There are still some PostScript printers, too.

Printing with Linux

Modern Linux systems employ two software suites to perform and manage printing. The first, CUPS (Common Unix Printing System), provides print drivers and print-job management; the second, Ghostscript, a PostScript interpreter, acts as a RIP.

CUPS manages printers by creating and maintaining print queues. As we discussed in our brief history lesson, Unix printing was originally designed to manage a centralized printer shared by multiple users. Since printers are slow by nature, compared to the computers that are feeding them, printing systems need a way to schedule multiple print jobs and keep things organized. CUPS also has the ability to recognize different types of data (within reason) and can convert files to a printable form.

Preparing Files for Printing

As command line users, we are mostly interested in printing text, though it is certainly possible to print other data formats as well.

pr—Convert Text Files for Printing

We looked at `pr` a little in the previous chapter. Now we will examine some of its many options used in conjunction with printing. In our history of printing, we saw that character-based printers use monospaced fonts, resulting in fixed numbers of characters per line and lines per page. pr is used to adjust text to fit on a specific page size, with optional page headers and margins. Table 22-1 summarizes the most commonly used options.

Table 22-1: Common pr Options

Option	Description
+*first*[:*last*]	Output a range of pages starting with *first* and, optionally, ending with *last*.
-*columns*	Organize the content of the page into the number of columns specified by *columns*.
-a	By default, multicolumn output is listed vertically. By adding the -a (across) option, content is listed horizontally.
-d	Double-space output.
-D *format*	Format the date displayed in page headers using *format*. See the man page for the date command for a description of the format string.
-f	Use form feeds rather than carriage returns to separate pages.
-h *header*	In the center portion of the page header, use *header* rather the name of the file being processed.
-l *length*	Set page length to *length*. Default is 66 lines (US letter at 6 lines per inch).
-n	Number lines.
-o *offset*	Create a left margin *offset* characters wide.
-w *width*	Set page width to *width*. Default is 72 characters.

pr is often used in pipelines as a filter. In this example, we will produce a directory listing of */usr/bin* and format it into paginated, three-column output using pr:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head


2012-02-18 14:00                                          Page 1
[                              apturl               bsd-write
411toppm                       ar                   bsh
a2p                            arecord              btcflash
a2ps                           arecordmidi          bug-buddy
a2ps-lpr-wrapper               ark                  buildhash
```

Sending a Print Job to a Printer

The CUPS printing suite supports two methods of printing historically used on Unix-like systems. One method, called Berkeley or LPD (used in the Berkeley Software Distribution version of Unix), uses the `lpr` program; the other method, called SysV (from the System V version of Unix), uses the `lp` program. Both programs do roughly the same thing. Choosing one over the other is a matter of personal taste.

lpr—Print Files (Berkeley Style)

The `lpr` program can be used to send files to the printer. It may also be used in pipelines, as it accepts standard input. For example, to print the results of our multicolumn directory listing above, we could do this:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

The report would be sent to the system's default printer. To send the file to a different printer, the `-P` option can used like this:

```
lpr -P printer_name
```

where *`printer_name`* is the name of the desired printer. To see a list of printers known to the system:

```
[me@linuxbox ~]$ lpstat -a
```

Note: *Many Linux distributions allow you to define a "printer" that outputs files in PDF, rather than printing on the physical printer. This is very handy for experimenting with printing commands. Check your printer configuration program to see if it supports this configuration. On some distributions, you may need to install additional packages (such as `cups-pdf`) to enable this capability.*

Table 22-2 shows some of the common options for `lpr`.

Table 22-2: Common lpr Options

Option	Description
`-#` *`number`*	Set number of copies to *`number`*.
`-p`	Print each page with a shaded header with the date, time, job name, and page number. This so-called "pretty print" option can be used when printing text files.
`-P` *`printer`*	Specify the name of the printer used for output. If no printer is specified, the system's default printer is used.
`-r`	Delete files after printing. This would be useful for programs that produce temporary printer-output files.

lp—Print Files (System V Style)

Like `lpr`, `lp` accepts either files or standard input for printing. It differs from `lpr` in that it supports a different (and slightly more sophisticated) option set. Table 22-3 lists the common options.

Table 22-3: Common lp Options

Option	Description
`-d` *`printer`*	Set the destination (printer) to *`printer`*. If no `d` option is specified, the system default printer is used.
`-n` *`number`*	Set the number of copies to *`number`*.
`-o landscape`	Set output to landscape orientation.
`-o fitplot`	Scale the file to fit the page. This is useful when printing images, such as JPEG files.
`-o scaling=`*`number`*	Scale file to *`number`*. The value of 100 fills the page. Values less than 100 are reduced, while values greater than 100 cause the file to be printed across multiple pages.
`-o cpi=`*`number`*	Set the output characters per inch to *`number`*. Default is 10.
`-o lpi=`*`number`*	Set the output lines per inch to *`number`*. Default is 6.

`-o page-bottom=`*`points`*
`-o page-left=`*`points`*
`-o page-right=`*`points`*
`-o page-top=`*`points`*
	Set the page margins. Values are expressed in *`points`*, a unit of typographic measurement. There are 72 points to an inch.

`-P` *`pages`*	Specify the list of pages. *`pages`* may be expressed as a comma-separated list and/or a range—for example `1,3,5,7-10`.

We'll produce our directory listing again, this time printing 12 CPI and 8 LPI with a left margin of one-half inch. Note that we have to adjust the `pr` options to account for the new page size:

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36 -o cpi=
12 -o lpi=8
```

This pipeline produces a four-column listing using smaller type than the default. The increased number of characters per inch allows us to fit more columns on the page.

Another Option: a2ps

The a2ps program is interesting. As we can surmise from its name, it's a format conversion program, but it's also much more. Its name originally meant *ASCII to PostScript*, and it was used to prepare text files for printing on PostScript printers. Over the years, however, the capabilities of the program have grown, and now its name means *Anything to PostScript*. While its name suggests a format-conversion program, it is actually a printing program. It sends its default output, rather than standard output, to the system's default printer. The program's default behavior is that of a "pretty printer," meaning that it improves the appearance of output. We can use the program to create a PostScript file on our desktop:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file `/home/me/Desktop/ls.ps'
```

Here we filter the stream with pr, using the -t option (omit headers and footers) and then, with a2ps, specifying an output file (-o option) and 66 lines per page (-L option) to match the output pagination of pr. If we view the resulting file with a suitable file viewer, we will see the output shown in Figure 22-1.

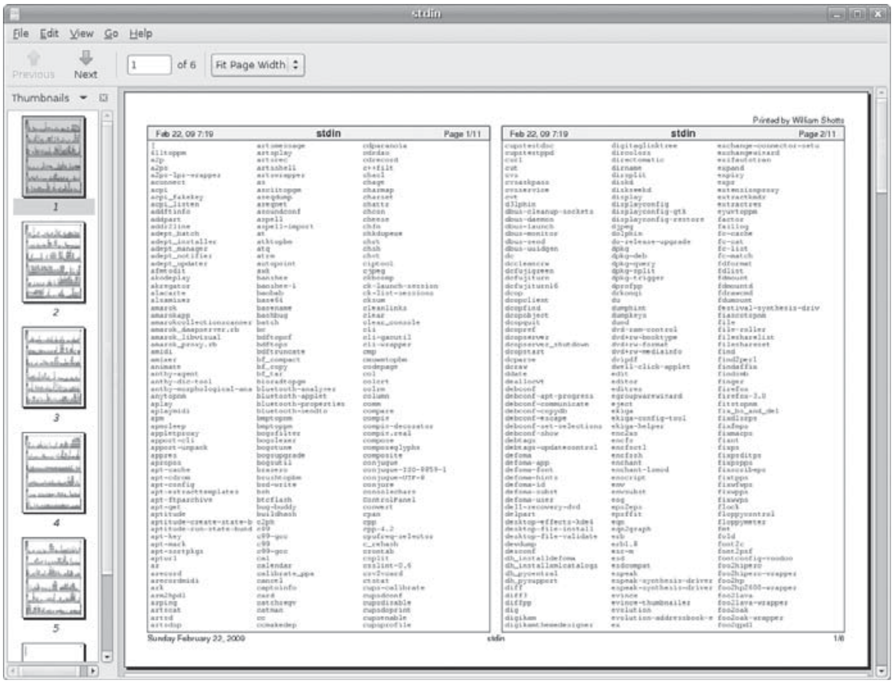


Figure 22-1: Viewing a2ps output

As we can see, the default output layout is "two up" format. This causes the contents of two pages to be printed on each sheet of paper. `a2ps` applies nice page headers and footers, too.

`a2ps` has a lot of options. Table 22-4 summarizes them.

Table 22-4: a2ps Options

Option	Description
`--center-title` *`text`*	Set center page title to *`text`*.
`--columns` *`number`*	Arrange pages into *`number`* columns. Default is 2.
`--footer` *`text`*	Set page footer to *`text`*.
`--guess`	Report the types of files given as arguments. Since `a2ps` tries to convert and format all types of data, this option can be useful for predicting what `a2ps` will do when given a particular file.
`--left-footer` *`text`*	Set left-page footer to *`text`*.
`--left-title` *`text`*	Set left-page title to *`text`*.
`--line-numbers=`*`interval`*	Number lines of output every *`interval`* lines.
`--list=defaults`	Display default settings.
`--list=`*`topic`*	Display settings for *`topic`*, where *`topic`* is one of the following: delegations (external programs that will be used to convert data), encodings, features, variables, media (paper sizes and the like), ppd (PostScript printer descriptions), printers, prologues (portions of code that are prefixed to normal output), stylesheets, or user options.
`--pages` *`range`*	Print pages in range.
`--right-footer` *`text`*	Set right-page footer to *`text`*.
`--right-title` *`text`*	Set right-page title to *`text`*.
`--rows` *`number`*	Arrange pages into *`number`* rows. Default is 1.
`-B`	No page headers.
`-b` *`text`*	Set page header to *`text`*.
`-f` *`size`*	Use *`size`* point font.
`-l` *`number`*	Set characters per line to *`number`*. This and the `-L` option (below) can be used to make files paginated with other programs, such as `pr`, fit correctly on the page.

(*continued*)

Table 22-4 (*continued*)

Option	Description
`-L` *`number`*	Set lines per page to *`number`*.
`-M` *`name`*	Use media name—for example, `A4`.
`-n` *`number`*	Output *`number`* copies of each page.
`-o` *`file`*	Send output to *`file`*. If *`file`* is specified as `-`, use standard output.
`-P` *`printer`*	Use *`printer`*. If a printer is not specified, the system default printer is used.
`-R`	Portrait orientation
`-r`	Landscape orientation
`-T` *`number`*	Set tab stops to every *`number`* characters.
`-u` *`text`*	Underlay (watermark) pages with *`text`*.

This is just a summary. `a2ps` has several more options.

Note: *`a2ps` is still in active development. During my testing, I noticed different behavior on various distributions. On CentOS 4, output always went to standard output by default. On CentOS 4 and Fedora 10, output defaulted to A4 media, despite the program being configured to use letter-size media by default. I could overcome these issues by explicitly specifying the desired option. On Ubuntu 8.04, `a2ps` performed as documented.*

Also note that there is another output formatter that is useful for converting text into PostScript. Called `enscript`, it can perform many of the same kinds of formatting and printing tricks, but unlike `a2ps`, it accepts only text input.

Monitoring and Controlling Print Jobs

As Unix printing systems are designed to handle multiple print jobs from multiple users, CUPS is designed to do the same. Each printer is given a *print queue*, where jobs are parked until they can be *spooled* to the printer. CUPS supplies several command-line programs that are used to manage printer status and print queues. Like the `lpr` and `lp` programs, these management programs are modeled after the corresponding programs from the Berkeley and System V printing systems.

lpstat—Display Print System Status

The `lpstat` program is useful for determining the names and availability of printers on the system. For example, if we had a system with both a physical printer (named *printer*) and a PDF virtual printer (named *PDF*), we could check their status like this:

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 05 Dec 2011 03:05:59 PM EST
printer accepting requests since Tue 21 Feb 2012 08:43:22 AM EST
```

Further, we could determine a more detailed description of the print system configuration this way:

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

In this example, we see that *printer* is the system's default printer and that it is a network printer using Internet Printing Protocol (*ipp://*) attached to a system named *print-server*.

The commonly used options are described in Table 22-5.

Table 22-5: Common lpstat Options

Option	Description
`-a [printer...]`	Display the state of the printer queue for *printer*. Note that this is the status of the printer queue's ability to accept jobs, not the status of the physical printers. If no printers are specified, all print queues are shown.
`-d`	Display the name of the system's default printer.
`-p [printer...]`	Display the status of the specified *printer*. If no printers are specified, all printers are shown.
`-r`	Display the status of the print server.
`-s`	Display a status summary.
`-t`	Display a complete status report.

lpq—Display Printer Queue Status

To see the status of a printer queue, the `lpq` program is used. This allows us to view the status of the queue and the print jobs it contains. Here is an example of an empty queue for a system default printer named *printer*:

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

If we do not specify a printer (using the -P option), the system's default printer is shown. If we send a job to the printer and then look at the queue, we will see it listed:

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
printer is ready and printing
Rank    Owner   Job     File(s)                         Total Size
active  me      603     (stdin)                         1024 bytes
```

lprm and cancel—Cancel Print Jobs

CUPS supplies two programs used to terminate print jobs and remove them from the print queue. One is Berkeley style (lprm), and the other is System V (cancel). They differ slightly in the options they support but do basically the same thing. Using our print job above as an example, we could stop the job and remove it this way:

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Each command has options for removing all the jobs belonging to a particular user, particular printer, and multiple job numbers. Their respective man pages have all the details.

23
COMPILING PROGRAMS

In this chapter, we will look at how to build programs by compiling source code. The availability of source code is the essential freedom that makes Linux possible. The entire ecosystem of Linux development relies on free exchange between developers. For many desktop users, compiling is a lost art. It used to be quite common, but today, distribution providers maintain huge repositories of precompiled binaries, ready to download and use. At the time of this writing, the Debian repository (one of the largest of any of the distributions) contains almost 23,000 packages.

So why compile software? There are two reasons:

- **Availability.** Despite the number of precompiled programs in distribution repositories, some distributions may not include all the desired applications. In this case, the only way to get the desired program is to compile it from source.
- **Timeliness.** While some distributions specialize in cutting-edge versions of programs, many do not. This means that in order to have the very latest version of a program, compiling is necessary.

Compiling software from source code can become very complex and technical, well beyond the reach of many users. However, many compiling tasks are quite easy and involve only a few steps. It all depends on the package. We will look at a very simple case in order to provide an overview of the process and as a starting point for those who wish to undertake further study.

We will introduce one new command:

- `make`—Utility to maintain programs.

What Is Compiling?

Simply put, compiling is the process of translating *source code* (the human-readable description of a program written by a programmer) into the native language of the computer's processor.

The computer's processor (or *CPU*) works at a very elemental level, executing programs in what is called *machine language*. This is a numeric code that describes very small operations, such as "add this byte," "point to this location in memory," or "copy this byte." Each of these instructions is expressed in binary (ones and zeros). The earliest computer programs were written using this numeric code, which may explain why programmers who wrote it were said to smoke a lot, drink gallons of coffee, and wear thick glasses.

This problem was overcome by the advent of *assembly language*, which replaced the numeric codes with (slightly) easier to use character *mnemonics* such as CPY (for copy) and MOV (for move). Programs written in assembly language are processed into machine language by a program called an *assembler*. Assembly language is still used today for certain specialized programming tasks, such as *device drivers* and *embedded systems*.

We next come to what are called *high-level programming languages*. They are called this because they allow the programmer to be less concerned with the details of what the processor is doing and more with solving the problem at hand. The early ones (developed during the 1950s) included *FORTRAN* (designed for scientific and technical tasks) and *COBOL* (designed for business applications). Both are still in limited use today.

While there are many popular programming languages, two predominate. Most programs written for modern systems are written in either C or C++. In the examples to follow, we will be compiling a C program.

Programs written in high-level programming languages are converted into machine language by processing them with another program, called a *compiler*. Some compilers translate high-level instructions into assembly language and then use an assembler to perform the final stage of translation into machine language.

A process often used in conjunction with compiling is called *linking*. Programs perform many common tasks. Take, for instance, opening a file.

Many programs perform this task, but it would be wasteful to have each program implement its own routine to open files. It makes more sense to have a single piece of programming that knows how to open files and to allow all programs that need it to share it. Providing support for common tasks is accomplished by what are called *libraries*. They contain multiple *routines*, each performing some common task that multiple programs can share. If we look in the */lib* and */usr/lib* directories, we can see where many of them live. A program called a *linker* is used to form the connections between the output of the compiler and the libraries that the compiled program requires. The final result of this process is the *executable program file*, ready for use.

Are All Programs Compiled?

No. As we have seen, some programs, such as shell scripts, do not require compiling but are executed directly. These are written in what are known as *scripting* or *interpreted* languages. These languages, which have grown in popularity in recent years, include Perl, Python, PHP, Ruby, and many others.

Scripted languages are executed by a special program called an *interpreter*. An interpreter inputs the program file and reads and executes each instruction contained within it. In general, interpreted programs execute much more slowly than compiled programs. This is because each source code instruction in an interpreted program is translated every time it is carried out, whereas with a compiled program, a source code instruction is translated only once, and this translation is permanently recorded in the final executable file.

So why are interpreted languages so popular? For many programming chores, the results are "fast enough," but the real advantage is that it is generally faster and easier to develop interpreted programs than compiled programs. Programs are usually developed in a repeating cycle of code, compile, test. As a program grows in size, the compilation phase of the cycle can become quite long. Interpreted languages remove the compilation step and thus speed up program development.

Compiling a C Program

Let's compile something. Before we do that, however, we're going to need some tools like the compiler, the linker, and `make`. The C compiler used almost universally in the Linux environment is called `gcc` (GNU C Compiler), originally written by Richard Stallman. Most distributions do not install `gcc` by default. We can check to see if the compiler is present like this:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

The results in this example indicate that the compiler is installed.

Note: *Your distribution may have a metapackage (a collection of packages) for software development. If so, consider installing it if you intend to compile programs on your system. If your system does not provide a metapackage, try installing the* `gcc` *and* `make` *packages. On many distributions, they are sufficient to carry out the exercise below.*

Obtaining the Source Code

For our compiling exercise, we are going to compile a program from the GNU Project called `diction`. This handy little program checks text files for writing quality and style. As programs go, it is fairly small and easy to build.

Following convention, we're first going to create a directory for our source code named *src* and then download the source code into it using `ftp`:

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--    1 1003  65534   68940 Aug 28  1998 diction-0.7.tar.gz
-rw-r--r--    1 1003  65534   90957 Mar 04  2002 diction-1.02.tar.gz
-rw-r--r--    1 1003  65534  141062 Sep 17  2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz (141062
bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

Note: *Since we are the maintainer of this source code while we compile it, we will keep it in* ~/src*. Source code installed by your distribution will be installed in* /usr/src*, while source code intended for use by multiple users is usually installed in* /usr/local/src.

As we can see, source code is usually supplied in the form of a compressed tar file. Sometimes called a *tarball*, this file contains the *source tree*, or hierarchy of directories and files that compose the source code. After arriving at the FTP site, we examine the list of tar files available and select the newest version for download. Using the `get` command within `ftp`, we copy the file from the FTP server to the local machine.

Once the tar file is downloaded, it must be unpacked. This is done with the tar program:

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11        diction-1.11.tar.gz
```

Note: *The* `diction` *program, like all GNU Project software, follows certain standards for source code packaging. Most other source code available in the Linux ecosystem also follows this standard. One element of the standard is that when the source code tar file is unpacked, a directory will be created that contains the source tree and that this directory will be named* project-x.xx, *thus containing both the project's name and its version number. This scheme allows easy installation of multiple versions of the same program. However, it is often a good idea to examine the layout of the tree before unpacking it. Some projects will not create the directory but instead will deliver the files directly into the current directory. This will make a mess in your otherwise well-organized* src *directory. To avoid this, use the following command to examine the contents of the tar file:*

```
tar tzvf tarfile | head
```

Examining the Source Tree

Unpacking the tar file results in the creation of a new directory, named *diction-1.11*. This directory contains the source tree. Let's look inside:

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess  diction.c        getopt.c      nl
config.h.in   diction.pot      getopt.h      nl.po
config.sub    diction.spec     getopt_int.h  README
configure     diction.spec.in  INSTALL       sentence.c
configure.in  diction.texi.in  install-sh    sentence.h
COPYING       en               Makefile.in   style.1.in
de            en_GB            misc.c        style.c
de.po         en_GB.po         misc.h        test
diction.1.in  getopt1.c        NEWS
```

In it, we see a number of files. Programs belonging to the GNU Project, as well as many others, will supply the documentation files *README*, *INSTALL*, *NEWS*, and *COPYING*. These files contain the description of the program, information on how to build and install it, and its licensing terms. It is always a good idea to carefully read the *README* and *INSTALL* files before attempting to build the program.

The other interesting files in this directory are the ones ending with *.c* and *.h*:

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c  getopt1.c  getopt.c  misc.c  sentence.c  style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h  getopt_int.h  misc.h  sentence.h
```

The *.c* files contain the two C programs supplied by the package (*style* and *diction*), divided into modules. It is common practice for large programs to be broken into smaller, easier-to-manage pieces. The source code files are ordinary text and can be examined with `less`:

```
[me@linuxbox diction-1.11]$ less diction.c
```

The *.h* files are known as *header files*. These, too, are ordinary text. Header files contain descriptions of the routines included in a source code file or library. In order for the compiler to connect the modules, it must receive a description of all the modules needed to complete the entire program. Near the beginning of the *diction.c* file, we see this line:

```
#include "getopt.h"
```

This instructs the compiler to read the file *getopt.h* as it reads the source code in *diction.c* in order to "know" what's in *getopt.c*. The *getopt.c* file supplies routines that are shared by both the `style` and `diction` programs.

Above the `include` statement for *getopt.h*, we see some other `include` statements such as these:

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

These also refer to header files, but they refer to header files that live outside the current source tree. They are supplied by the system to support the compilation of every program. If we look in */usr/include*, we can see them:

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

The header files in this directory were installed when we installed the compiler.

Building the Program

Most programs build with a simple, two-command sequence:

```
./configure
make
```

The `configure` program is a shell script that is supplied with the source tree. Its job is to analyze the build environment. Most source code is designed to be *portable*. That is, it is designed to build on more than one kind of Unix-like system. But in order to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. `configure` also checks to see that necessary external tools and components are installed.

Let's run configure. Since configure is not located where the shell normally expects programs to be located, we must explicitly tell the shell its location by prefixing the command with ./. This indicates that the program is located in the current working directory:

```
[me@linuxbox diction-1.11]$ ./configure
```

configure will output a lot of messages as it tests and configures the build. When it finishes, the output will look something like this:

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

What's important here is that there are no error messages. If there were, the configuration would have failed, and the program would not build until the errors are corrected.

We see configure created several new files in our source directory. The most important one is *Makefile*. *Makefile* is a configuration file that instructs the make program exactly how to build the program. Without it, make will refuse to run. *Makefile* is an ordinary text file, so we can view it:

```
[me@linuxbox diction-1.11]$ less Makefile
```

The make program takes as input a *makefile* (which is normally named *Makefile*), which describes the relationships and dependencies among the components that compose the finished program.

The first part of the makefile defines variables that are substituted in later sections of the makefile. For example, we see the line

```
CC=                gcc
```

which defines the C compiler to be gcc. Later in the makefile, we see one instance where it gets used:

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

A substitution is performed here, and the value $(CC) is replaced by gcc at runtime.

Most of the makefile consists of lines, which define a *target*—in this case the executable file *diction*—and the files on which it is dependent. The remaining lines describe the command(s) needed to create the target from its components. We see in this example that the executable file *diction* (one of the final end products) depends on the existence of *diction.o*, *sentence.o*, *misc.o*, *getopt.o*, and *getopt1.o*. Later on, in the makefile, we see definitions of each of these as targets.

```
diction.o:       diction.c config.h getopt.h misc.h sentence.h
getopt.o:        getopt.c getopt.h getopt_int.h
getopt1.o:       getopt1.c getopt.h getopt_int.h
misc.o:          misc.c config.h misc.h
sentence.o:      sentence.c config.h misc.h sentence.h
style.o:         style.c config.h getopt.h misc.h sentence.h
```

However, we don't see any command specified for them. This is handled by a general target, earlier in the file, that describes the command used to compile any *.c* file into a *.o* file:

```
.c.o:
                 $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer will become clear in a moment. In the meantime, let's run `make` and build our programs:

```
[me@linuxbox diction-1.11]$ make
```

The `make` program will run, using the contents of *Makefile* to guide its actions. It will produce a lot of messages.

When it finishes, we will see that all the targets are now present in our directory:

```
[me@linuxbox diction-1.11]$  ls
config.guess   de.po            en            install-sh   sentence.c
config.h       diction          en_GB         Makefile     sentence.h
config.h.in    diction.1        en_GB.mo      Makefile.in  sentence.o
config.log     diction.1.in     en_GB.po      misc.c       style
config.status  diction.c        getopt1.c     misc.h       style.1
config.sub     diction.o        getopt1.o     misc.o       style.1.in
configure      diction.pot      getopt.c      NEWS         style.c
configure.in   diction.spec     getopt.h      nl           style.o
COPYING        diction.spec.in  getopt_int.h  nl.mo        test
de             diction.texi     getopt.o      nl.po
de.mo          diction.texi.in  INSTALL       README
```

Among the files, we see `diction` and `style`, the programs that we set out to build. Congratulations are in order! We just compiled our first programs from source code!

But just out of curiosity, let's run `make` again:

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

It produces only this strange message. What's going on? Why didn't it build the program again? Ah, this is the magic of make. Rather than simply build everything again, make builds only what needs building. With all of the targets present, make determined that there was nothing to do. We can demonstrate this by deleting one of the targets and running make again to see what it does.

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

We see that make rebuilds *getopt.o* and relinks the diction and style programs, since they depend on the missing module. This behavior also points out another important feature of make: It keeps targets up-to-date. make insists that targets be newer than their dependencies. This makes perfect sense, as a programmer will often update a bit of source code and then use make to build a new version of the finished product. make ensures that everything that needs building based on the updated code is built. If we use the touch program to "update" one of the source code files, we can see this happen:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

After make runs, we see that it has restored the target to being newer than the dependency:

```
[me@linuxbox diction-1.11]$  ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

The ability of make to intelligently build only what needs building is a great benefit to programmers. While the time savings may not be apparent with our small project, it is significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) contains several *million* lines of code.

Installing the Program

Well-packaged source code often includes a special make target called install. This target will install the final product in a system directory for use. Usually, this directory is */usr/local/bin*, the traditional location for locally built software. However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation:

```
[me@linuxbox diction-1.11]$ sudo make install
```

After we perform the installation, we can check that the program is ready to go:

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

And there we have it!

Final Note

In this chapter, we have seen how three simple commands—`./configure`, `make`, `make install`—can be used to build many source code packages. We have also seen the important role that `make` plays in the maintenance of programs. The `make` program can be used for any task that needs to maintain a target/dependency relationship, not just for compiling source code.