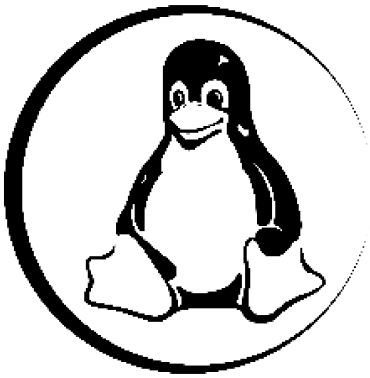


Using Command-Line Tools

Because Linux has its conceptual roots in the older Unix operating system, many Linux applications are designed to be executed from a command line. A command line is the traditional interface found on older computer systems that may not have used the high-resolution, graphically oriented monitors that most people expect today. In the command-line model, the system runs a program, known as a *command-line interpreter*, which does just what its name suggests. A command-line interpreter reads the commands that you type, locates the appropriate application on your system, and executes that application for you as instructed based on what you've typed. Once the command completes, the command interpreter displays a sequence of characters, known as a *prompt*, signifying that it is ready to accept another command.

Linux systems aren't addicted to the command-line approach simply out of historical interest — when a Linux system boots, it needs to be able to run several commands long before the graphical interface is available. On Linux systems, the graphical interface itself is started by a command-line utility, after which (of course), fancy graphics are available, expected, and used.

For most people, simply displaying a command-line prompt on a fancy graphical monitor powered by a multi-megabyte graphics card would be a waste of good hardware. However, Linux provides thousands of useful and extremely powerful command-line applications, so graphical user interfaces (GUIs) such as GNOME and the X Window system offer applications that display a command prompt in a separate window, giving you the best of both worlds. You can be running Open Office Writer or the GIMP in separate windows, making the most of their graphical capabilities, while you can be simultaneously displaying a command prompt in another and executing command-line utilities there.

This chapter explains the basic ideas behind command-line utilities and the commands that execute them, discusses the primary Linux command-line interpreters, explains how to access them, and explores some of the most popular utilities for running command-line utilities on your Ubuntu Linux system.

NOTE **Most of the sample command-line examples given throughout this chapter begin with the character `$`, which is the default prompt used by Ubuntu's default Linux command interpreter. (A prompt is the character or characters that a command interpreter displays to signify that it is waiting for you to type something.) If you're following along, you should not type this character — you'll already see it (or some more complex prompt) at the beginning of your command line.**

Why Use the Command Line?

Linux provides thousands of command-line utilities that range from simple programs for creating, examining, and modifying files and file permissions to complex utilities that enable you to fine-tune the performance of your hardware and low-level operating system capabilities such as filesystems and networking. Many of these applications have graphical equivalents, especially user-level applications and system configuration utilities. For example, the System ➪ Administration menu's Networking menu item starts the graphical `network-admin` application, which performs the same functions as the command-line `ifconfig` utility. Similarly, GNOME's Nautilus file manager is roughly equivalent to a Linux command-line interpreter in general, because it enables you to examine and manipulate files and directories, execute other applications, and so on.

Using the command line isn't really an alien notion even to users of operating systems such as Microsoft Windows and Mac OS X (though it's newer to the latter). Many Windows users, especially system administrators, have always found a certain convenience in some of the command-line configuration utilities provided with Windows systems, especially those related to network configuration and status reporting such as `ipconfig`. The idea of a command line may be somewhat new to long-time Macintosh users, because the command line was a serious alternative to the GUI when Apple introduced its Unix-based Mac OS X operating system.

Even if you don't specifically want to use the Linux command line, there is one case in which you may have to, which is if your Ubuntu system encounters major hardware or configuration problems during the boot process. If, for example, your system finds that its root filesystem is corrupted or inaccessible during the boot process, you'll see a command-line prompt faster than you can say, "Hmmm, I wonder if that's a major problem." All Linux systems fall back to the command line when major problems are identified during the boot sequence to give you access to the command-line applications that you'll need to use in order to correct the problem and reenable the system to boot normally. The Linux boot process is a command-line process, executing the appropriate configuration utilities in the correct order, the last of which is the command that starts the X Window system and your GNOME desktop (or window manager, if you've customized things). Even if your system's filesystems are in good shape and all of your hardware and associated system software runs correctly during the boot process, configuration problems with the X Window system are a common cause of seeing a command-line prompt instead of the glorious Ubuntu login screen when you boot your system. The Ubuntu Linux boot process is described in detail in Chapter 19, "Understanding the Ubuntu Startup and Shutdown Processes."

Executing Commands from the Command Line

Applications that are designed to be executed from a Linux command line are typically referred to as command-line utilities. All command-line utilities have the same general organization — they begin with the name of the command that you want to execute, and are optionally followed by information about the way in which you want the command to behave. Anything following the name of the command that you

want to execute is known as an *argument* to that command. The arguments to each command-line utility differ based on the command that you want to execute. In an interesting example of recursion, the arguments to command-line utilities are generally referred to as *options*, each of which may take an argument.

Confused? If this is all new to you, don't worry. A few examples will help clear this up, and soon you'll be as comfortable at the command line as anyone. Let's use the Linux `ls` command as an example, because it has more options than almost any other command and was also introduced in the section of Chapter 4 entitled "Understanding Linux Permissions."

The `ls` command lists information about files and directories on your Ubuntu system. The Linux `ls` command is an updated version of a classic Unix utility by the same name. In true Unix fashion, no one was willing to type extra characters like "i" and "t", so the command was given the faster-to-type abbreviation of `ls`.

When used by itself on the command line, the `ls` command simply displays the contents of the current directory, as in the following example:

```
$ ls
boot_services.txt  hello.c    hello.o  include_example.c
include_test
hello              hello.foo  hello.s  include_example.out
```

You can also supply the name of a specific file or directory as an argument to the `ls` command, as in the following examples:

```
$ ls hello
hello
```

Using the `ls` command to list the name of a file that you already know is spectacularly uninteresting (though it can be very useful when combined with wildcards, which are discussed later in this chapter). However, listing a directory shows the contents of that directory, as in the following example:

```
$ ls include_test
libxml2  netdev  system
```

The output from this command shows that the `include_test` directory itself contains three other files or directories. I happened to know that `include_test` was a directory — if you're not sure what types of things are in the current directory, you can use the `ls` command's `-F` option to give you this information. For example, here's the current directory as shown using the `ls -F` command:

```
$ ls -F
boot_services.txt  hello.c    hello.o  include_example.c
include_test/
hello*             hello.foo  hello.s  include_example.out
```

The `ls` command's `-F` option decorates the names of the objects in the current directory with an extra character to identify any object that isn't simply a text file. An asterisk following the name of an object shows that this is an executable file, while a slash ("`/`") following the name of an object shows that this is indeed a directory.

The `ls` command's `-F` option is very useful, but (in true Linux fashion) isn't the only way to get detailed information about each of the objects in the current directory. You can also get this sort of information using other options to the `ls` command. For example, one of the most commonly used options to the `ls` command is the `-l` option, which means "display output in long format." Using this option gives a variety of additional information about the objects in the current directory, as in the following example:

```
$ ls -l
total 44
-rw-r--r--  1 wvh users  783 2006.03-15 06:36 boot_services.txt
-rwxr-xr-x  1 wvh users 9249 2006.03-15 06:37 hello
-rw-r--r--  1 wvh users   60 2006.03-15 06:37 hello.c
-rw-r--r--  1 wvh users   60 2006.03-15 06:37 hello.foo
-rw-r--r--  1 wvh users 2504 2006.03-15 06:37 hello.o
-rw-r--r--  1 wvh users  857 2006.03-15 06:37 hello.s
-rw-r--r--  1 wvh users  202 2006.03-15 06:37 include_example.c
-rw-r--r--  1 wvh users  736 2006.03-15 06:37 include_example.out
drwxr-xr-x  5 wvh users 4096 2006.03-15 06:37 include_test
```

As you can see, the long option displays more complete information about the files and directories in the current directory. From left to right, this information consists of the following: current permissions, the number of hard links to that file in the Linux filesystem (more about that later in this section), the owner and group, size, the date and time at which it was last modified, and the file or directory name.

As mentioned previously, you can combine options and arguments on the same command line to refine the behavior of most command-line utilities. For example, to get a long listing of the contents of the `include_test` directory, you would execute the following command:

```
$ ls -l include_test
total 0
drwxr-xr-x  2 wvh users 72 2006.03-15 06:37 libxml2
drwxr-xr-x  2 wvh users 80 2006.03-15 06:37 netdev
drwxr-xr-x  4 wvh users 96 2006.03-15 06:37 system
```

One other very popular option to the `ls` command is the `-a` option, which shows all of the objects in the current directory. By default, the `ls` command doesn't show objects whose names begin with a period (aka full stop). This is because all Linux directories contain two special entries that many people don't care about, but which are useful to traverse and support the hierarchical structure of a Linux filesystem. These are the "`.`" entry, which always refers to the current directory, and the "`..`" entry, which always refers to the parent of the current directory. Using the `ls -a` command to look at the contents of the current directory shows the following:

```
$ ls -a
.                   hello.c     include_example.c     .run_me_now
..                  hello.foo   include_example.out
boot_services.txt   hello.o     include_test
hello               hello.s     .my_music_directory
```

You'll note that the "`.`" and "`..`" entries are listed in the first column. However, you'll also note that two new files have appeared in the directory. These are the files `.my_music_directory` and `.run_me_now`, which are listed in the directory listing based on the first alphanumeric character in their names because the `ls` command ignores leading periods when sorting file names (unless, of course, the filename has no other characters as is the case with the "`.` " and "`..` " entries, which therefore appear first in the listing).

TIP **Because files beginning with periods aren't included in directory listings unless you use the `-a` option, using file names that begin with a period has become a standard convention for "hiding" those files. Files and directories whose names begin with a period are most commonly used to hold configuration information used by various Linux commands. The most common example of this type of file is the `.bashrc` configuration file used by the bash shell, which I'll discuss later in this chapter.**

You can combine multiple single-letter options after a single leading dash, and the `ls` command will perform all of the specified actions. For example, let's combine the `-a` and `-F` options to look at the current directory:

```
$ ls -aF
./                  hello.c     include_example.c     .run_me_now*
../                 hello.foo   include_example.out
boot_services.txt   hello.o     include_test/
hello*              hello.s     .my_music_directory@
```

From the output of this command, you can see that the "`.`" and "`..` " entries are both directories (actually, they are hard links to the current and parent directories), and the mysterious `.run_me_now` file is actually an executable command. The name of the `.my_music_directory` file is followed by an at symbol (`@`). What's up with that?

I've used the term "links" in passing previously, and this is a good time to explain it. Links are essentially just pointers to other files and directories in the Linux filesystem. Linux supports two types of links: hard links, which are actual connections to an existing file or directory, and symbolic links, which contain the name of some other file or directory that you want to refer to. Because hard links are actual connections to an existing data structure, the thing that you're linking to must actually exist and must be within the same disk partition as the thing that you're hard linking to. (This is a side effect of the internal data structures used by filesystems, which are too detailed to discuss here. This whole options concept may already be making you drowsy, let alone plunging into a discussion of filesystem data structures.) Unlike hard links, because symbolic links just contain the name of something else, they can point to any file or directory on your machine. As you may now suspect, an `@` symbol following the name of a file or directory indicates that this file or directory is just a symbolic link to another file or directory somewhere else on your system. Let's combine the `ls` command's `-a` and `-l` options to get some detailed information about everything in the current directory:

```
$ ls -al
total 45
drwxr-xr-x  3 wvh users  400 2006.03-16 06:20 .
drwxr-xr-x 13 wvh users 1376 2006.03-16 05:38 ..
-rw-r—r—  1 wvh users  783 2006.03-15 06:36 boot_services.txt
-rwxr-xr-x  1 wvh users 9249 2006.03-15 06:37 hello
-rw-r—r—  1 wvh users   60 2006.03-15 06:37 hello.c
-rw-r—r—  1 wvh users   60 2006.03-15 06:37 hello.foo
-rw-r—r—  1 wvh users 2504 2006.03-15 06:37 hello.o
-rw-r—r—  1 wvh users  857 2006.03-15 06:37 hello.s
-rw-r—r—  1 wvh users  202 2006.03-15 06:37 include_example.c
-rw-r—r—  1 wvh users  736 2006.03-15 06:37 include_example.out
drwxr-xr-x  5 wvh users  120 2006.03-15 06:37 include_test
lrwxrwxrwx  1 wvh users   11 2006.03-16 06:20 .my_music -> /opt2/music
-rwxr-xr-x  1 wvh users  269 2006.03-16 05:47 .run_me_now
```

This actually shows you a fair amount of information about everything in this directory, especially in terms of links. Remember that the second column in a long directory listing identifies the number of hard links to each object. For example, the second column for the "`.`" entry shows that there are three hard links to the current directory—these are the "`.` " itself, it's entry in the parent directory, and the hard link to "`.`" that is

present in the `include_test` subdirectory because it is the parent of the `include-test` directory. The "`..`" entry, which is a hard link to the parent directory of the current directory, seems to be very popular because there are 13 hard links to it. This means that the parent directory of the current directory probably contains several other directories. Looking at the `.my_music`, you can see that it is indeed a symbolic link, because its name in the long listing actually shows the other object that it is a symbolic link to. In this case, the file `.my_music` is a symbolic link to a directory that happens to live on another filesystem that is mounted on the `/opt2/music` directory on my system.

All of the command-line options I've discussed up to this point have begun with a single dash. This isn't always necessarily the case for all commands and their options. Some Linux commands with their conceptual roots in ancient versions of Unix (like the `tar` command) don't even require a leading dash before a single command-line option or a single group of command-line options. This antique command-line convention is deprecated, which means that anyone who implements a command nowadays that doesn't require at least one dash before its options will be mocked and verbally abused via e-mail by the Linux and open source communities. (All Unix commands that don't require a dash before their options have also been updated so that they can also handle finding a dash before their options.) Nowadays, command-line options always begin with a dash, but in an interesting usability twist, they can also begin with two dashes. The conventions for this are that single-letter options are preceded by a single dash, while multi-letter, "whole word" options begin with two dashes. This is necessary for two reasons:

- Most command-line utilities support both styles of options: the traditional single letter options and the newer whole-word options pioneered by the folks at the Free Software Foundation.
- Because you can combine multiple, single-letter options into a single group of options (as you've seen throughout this section), unless you use two dashes to identify a whole-word option, the command that you're executing can't differentiate between the two.

To illustrate this, let's ask for help from the `ls` command:

```
$ ls -help
/bin/ls: invalid option—e
Try `/bin/ls —help' for more information.
```

Well, that was actually both illustrative and friendly. When preceded with a single dash, the `ls` command interprets each of the letters that follow as a single option, and therefore complains because no option matches the letter `e` in the `-help` option. Let's try that again, correctly. Running the `ls —help` command displays the following output (truncated here because this is just an example):

```
$ ls —help
Usage: /bin/ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuSUX 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 ..
[remaining output deleted]
```

Getting Information about Commands

As you can see from the final example in this section, many command-line utilities provide a `--help` option, which displays what is known as a usage message. A usage message is a short summary of how to use a command, summarizing available options and their meanings. Unfortunately, not all commands offer a `--help` option and those that do can't display all possible information about those options. As discussed in Chapter 5, the graphical user environment used on Ubuntu systems provides a huge assortment of online help that makes it easy for you to get information about how to use graphical commands. Luckily, the Ubuntu command-line environment also provides a similar amount of online help in the form of the `man` and `info` commands.

The `man` command is your best friend when using or simply experimenting with command-line programs. The `man` command displays online manual pages, formatted for your screen. In true Linux/Unix fashion, the `man` command paginates its output using the Linux version of the familiar Unix `more` command — known as `less` — to make it easy to scroll forward or backward in its output. Because the online reference information displayed by the `man` command is displayed a screen/page at a time and corresponds to the documentation you'd traditionally find in a printed reference manual, the documentation displayed by `man` is generally referred to as a *man page* or as *man pages*.

As an example of using the `man` command, you can type `man man` for additional information on the `man` command itself. The first part of this man page looks like the following:

```
man(1)                        Manual pager utils                        man(1)

NAME
       man - an interface to the on-line reference manuals

 SYNOPSIS
       man  [-c|-w|-tZHT  device]  [-adhu7V]  [-i|-I] [-m system[,...]] [-L
       locale] [-p string] [-M path] [-P pager] [-r prompt] [-S  list]  [-e
       extension] [[section] page ...] ...
       man  -l  [-7] [-tZHT device] [-p string] [-P pager] [-r prompt] file
       ...
       man -k [apropos options] regexp ...
       man -f [whatis options] page ...

DESCRIPTION
       man is the system's manual pager. Each page argument given to man is
       normally  the  name  of  a program, utility or function.  The manual
       page associated with each of these arguments is then found and  dis-
       played. A section, if provided, will direct man to look only in that
       section of the manual.  The default action is to search  in  all  of
       the  available  sections,  following a predefined order and to show
       only the first page found, even if page exists in several  sections.
[Additional output removed]
```

continued

continued
As you can see, the online man pages for Linux command provide extensive information about available options, but also provide a significant amount of general information about using the commands themselves, usually including examples.

A more modern alternative to the man command, the `info` command provides similar usage and explanatory information about Linux commands. The `info` command uses emacs to display its output and also provides more extensive and up-to-date command documentation than the traditional `man` command. The following figure shows a window displaying info on the `info` command.

The info command running in a terminal window

The `info` command was introduced by the GNU folks to provide a richer, more robust environment for displaying online help for command-line utilities. The text files that are used by the `info` command provide a very rich syntax for hyperlinking between different sections of info documentation, and the use of emacs as the application for displaying info pages provides a much more flexible environment for moving around in the info that you're looking at. The `info` command can also display traditional Linux `man` pages, so you can even examine existing man pages using `info` — the `info man` command is perhaps the best example of that sort of recursion.

The Linux `man` and `info` commands will quickly become your new best friends when you are using or simply learning about command-line utilities.

What's a Shell?

A shell is the generic name given to any Linux command-line interpreter, and comes from Unix, the conceptual parent of Linux. Unix was one of the first operating systems to introduce the idea of using a command-line interpreter that was not built into the operating system and which had no special permissions to do mysterious operating system tasks. These ideas have been preserved in every conceptual descendant of Unix and have proved handy for several reasons. The most interesting of these is that, because it is a separate,

stand-alone executable, a Linux system can offer each user their choice of multiple shells, all of which can be upgraded independently from the operating system.

The original Unix shell was written by Ken Thompson, one of the two primary creators of Unix, and was extended by John Mashey, also at Bell Labs. These shells were somewhat primitive — the first shell that has the types of capabilities we have come to know and love today was the Bourne Shell, written by Stephen Bourne at Bell Labs in 1974. This shell, known to its friends as `/bin/sh`, is the shell that is used by default on most Unix systems, and is the conceptual parent of the `/bin/bash` shell (which stands for Bourne-Again Shell) that is used by default on most Linux systems today. As I'll explain in the next section, there are a variety of other shells to choose from on Linux systems. If you've come to Linux from a version of Unix that features another shell, you'll be able to feel at home pretty quickly (and perhaps even bring over and reuse your existing shell configuration files.)

TIP **One term that you will frequently hear when discussing shells, Linux, or Unix in general is the term *shell script*. A shell script is a command file containing commands that are either internal to the shell (such as those for setting variables, conditional expressions, and looping constructs) or which reference other commands on your Linux system. The ability to write very sophisticated command files is a feature of the Linux/Unix utility model, where each utility does one thing and does it well, and many different utilities can be linked together via pipes or temporary files to process each other's output.**

For additional information about shells on Unix systems, see:

- `www.softpanorama.org/People/Shell_giants/introduction.shtml`
- `www.faqs.org/faqs/unix-faq/shell/shell-differences/`
- `www.unix.org.ua/orelly/unix/ksh/ch01_03.htm`

Available Shells for Linux Systems

A standard Ubuntu distribution installs only the standard GNU bash shell. However, other shells are available for installation through `apt-get` or the Synaptic Package Manager. The following shells are found on many Linux systems:

`/bin/ash`: The Almquist shell, a small-footprint shell that began as a clone of the SYSV R4 version of the Bourne shell. The ash shell is often used on embedded systems or during the system startup and installation process on some Linux distributions.

`/bin/bash`: The default, Bourne-Again shell inspired by `/bin/sh` and `/bin/ksh`. This is the default shell installed and used by all users on Ubuntu systems.

`/bin/csh`: If you install `tcsh` (see its description later in this list), a symbolic link from `/bin/csh` to `/bin/tcsh` is also created. This link is provided to support shell scripts that reference the traditional location of the C-Shell, the standard shell used on Berkeley Unix (BSD) systems and their descendants. The C-shell supports configuration commands that are reminiscent of C-language programming constructs, and was originally written by Bill Joy, later a founder of Sun Microsystems.

`/bin/dash`: The Debian Almquist shell, the Debian Linux distribution's version of the standard Almquist shell. If you're interested in experimenting with this shell, it is available in the standard Ubuntu repository.

continued

continued

`/bin/ksh`: If you install `pdksh` (see its description later in this list), a symbolic link from `/bin/ksh` to `/bin/pdksh` is also created. This link is provided to support shell scripts that reference the traditional location of the Korn shell, the standard shell used on later SYSV Unix systems from AT&T and their descendants

`/bin/nash`: Another small-footprint shell used during the startup process on many Linux systems, specifically on Red Hat and Fedora Core Linux systems

`/bin/pdksh`: An open source version of David Korn's Korn shell, written at AT&T and was available with the SYSV R3 (as part of the "Experimental Toolchest") and R4 (as a completely supported utility) Unix distributions. The Korn shell (and thus `pdksh`) is completely backward compatible with the original Bourne shell. If you're interested in experimenting with this shell, it is available in the standard Ubuntu repository.

`/bin/sh`: A symbolic link to `/bin/bash`, provided for compatibility with generic Linux and Unix shell scripts.

`/bin/tcsh`: The TENEX C-Shell, which is an advanced, open source version of the C-shell with command-line editing extensions that were originally introduced by the command interpreter used by DEC's TOPS-20 operating system for PDP-10 systems, which began life as BBN's TENEX (for the 10 in PDP-10) operating system and was therefore later mutated into TWENEX (for the 20 in TOPS-20) by the PDP-10 hacker community. If you're interested in experimenting with this shell, it is available in the standard Ubuntu repository.

`/bin/zsh`: The Z shell is a powerful, tremendously extensible shell that provides many of the capabilities of bash and ksh, and much, much more. If you're interested in experimenting with this shell, it is available in the standard Ubuntu repository.

You can use the `chsh` command-line command to change the default shell used by your account if you decide that you want to use a shell other than the one that is currently listed in your `/etc/passwd` entry. However, you can't change your login shell to any random binary — all of the programs to which you can change your shell are listed in the text file `/etc/shells` on your Ubuntu system. If you want to be able to set some other application as a login shell using the `chsh` command, you must first add it to the file `/etc/shells`.

Getting to a Shell

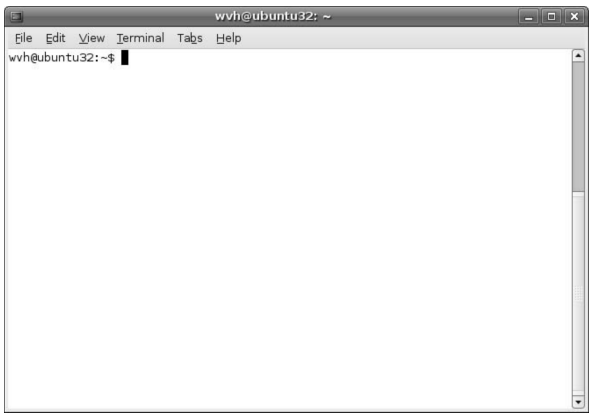
Assuming that I've hyped the value of the Linux command line sufficiently and that your Ubuntu system boots correctly in graphical mode, you're probably wondering how to start a shell so that you can experiment a bit. Ubuntu provides two applications that are the most common mechanisms for starting a shell, one that is a standard part of the X Window System distribution, and another that is a standard part of any GNOME distribution. Both of these are referred to as terminal applications, not because they imply an end to life as we know it, but because they are reminiscent of the user experience on systems without bitmapped graphical displays, when most users accessed their computer systems through terminals that couldn't do much more than display the input and output of a command interpreter.

Using the GNOME Terminal Application

The GNOME terminal application is the most common way of starting a command-line shell on a graphical Ubuntu system. To start the GNOME Terminal, select the Terminal command from the Applications ➪ Accessories menu. Figure 6.1 shows the default GNOME Terminal window.

FIGURE 6.1

The GNOME Terminal application

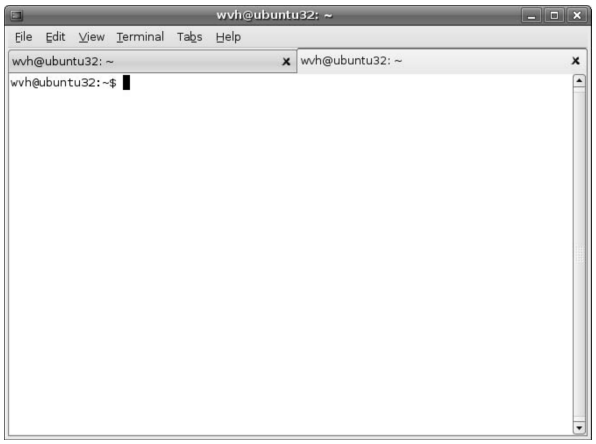


Like most GNOME applications, the GNOME Terminal provides extensive online help that is available by selecting Help ➪ Contents.

The GNOME Terminal application provides a variety of menus that make it easy to configure things like the title of a GNOME Terminal window (Terminal ➪ Set Title); configure the character set (Terminal ➪ Set Character Encoding); and configure the size, fonts, colors, and other display attributes of the GNOME Terminal window (Edit ➪ Current Profile). In my opinion, its most generally useful feature is its ability to open and manage multiple command-line sessions within a single Terminal window. To open a new tab, select the File ➪ New Tab command. A new tab displays, as shown in Figure 6.2.

FIGURE 6.2

Multiple tabs in the GNOME Terminal



Pipes

A *pipe* is an operator that combines input and output redirection so that the output of one command is immediately used as the input for another. The pipe is represented by the vertical line character (|), which is usually a shift character located somewhere near the Return or Enter key on your keyboard. Suppose you want to list the contents of a directory, but the directory's listing is so long that many of the entries scrolled off the screen before you can read them. A pipe gives you a simple way to display the output one page at a time, with the following command:

```
$ ls -l /etc | more
total 1780
-rw-r--r--    1 root     root        15221 Feb 28  2001 a2ps.cfg
-rw-r--r--    1 root     root         2561 Feb 28  2001 a2ps-site.cfg
-rw-r--r--    1 root     root           47 Dec 28  2001 adjtime
drwxr-xr-x    4 root     root         4096 Oct  1  2001 alchemist
-rw-r--r--    1 root     root         1048 Mar  3  2001 aliases
-rw-r--r--    1 root     root        12288 Sep  8  2003 aliases.db
-rw-r--r--    1 root     root          370 Apr  3  2001 anacrontab
-rw-------    1 root     root            1 Apr  4  2001 at.deny
-rw-r--r--    1 root     root          210 Mar  3  2001 auto.master
-rw-r--r--    1 root     root          574 Mar  3  2001 auto.misc
-rw-r--r--    1 root     root          823 Feb 28  2001 bashrc
drwxr-xr-x    3 root     root         4096 Apr  7  2001 CORBA
drwxr-xr-x    2 root     root         4096 Mar  8  2001 cron.d
drwxr-xr-x    2 root     root         4096 Oct  1  2001 cron.daily
drwxr-xr-x    2 root     root         4096 Oct  1  2001 cron.hourly
drwxr-xr-x    2 root     root         4096 Oct  1  2001 cron.monthly
-rw-r--r--    1 root     root          255 Feb 27  2001 crontab
drwxr-xr-x    2 root     root         4096 Oct  1  2001 cron.weekly
-rw-r--r--    1 root     root          380 Jul 25  2000 csh.cshrc
-rw-r--r--    1 root     root          517 Mar 27  2001 csh.login
drwxr-x---    2 root     root         4096 Oct  1  2001 default
--More--
```

Here's the same output from `ls -l /etc` that you saw earlier in this chapter, but this time it's limited to a single screen's worth of files because the output is piped through the `more` command. (Learn more about `more` in the "Common File Manipulation Commands" section later in this chapter.)

Pipes, redirection, and all the other features in this section can be combined in near-infinite combinations to create complex chains of commands. For example, the command

```
sort < terms > terms-alpha | mail fred
```

would perform the sort operation described previously, and then mail the contents of the `terms-alpha` *file to a user named fred. The only limitation to these combinations is the user's ingenuity.*

Command Substitution

Command substitution is yet another way of using the output of one command as an argument to another. It is a more complex operation than simply piping the output through a second command, but it can be used to create command strings of some sophistication. Consider the command:

```
ls $(pwd)
```

In this case, the command `pwd` is the first command to run; it outputs the name of the working directory. Its value is sent as an argument for the `ls` command. The output returned by this command would be the directory listing for the current working directory.

> *The astute reader will note that this example has much the same effect as the* `ls` *command itself, but it is a useful illustration of command substitution in operation.*

Yes, this command has much the same effect as:

```
pwd | ls
```

The output is the same, but there are some differences in the behind-the-scenes way that it's carried out. The construction using the `$` operator is distinctive in that the command in parentheses is executed in a *subshell*—that is, a new instance of the shell is spawned, the command is evaluated, the subshell closes, and the result is returned to the original shell.

> *If you have special environment conditions set up that might not transfer to a subshell (a manually set* `PATH` *value, for example), the subshell might not inherit this condition. In such a case, the command may fail.*

Instead of using the `$()` construction, you can also use backticks. For example,

```
ls `pwd`
```

accomplishes exactly the same thing as `ls $(pwd)`. Another alternative is to use curly braces, as in:

```
ls ${pwd}
```

The difference here is that the command in the curly braces is executed in the current shell, and no subshell process is spawned. (Curly braces do not work on all Unix variants.)

Working with Files and Directories

The most common Unix commands are those used to manage files and directories. You might find yourself issuing one of these commands hundreds of times a day as you go about your business. In this section, you learn the two most popular file management commands and the various arguments that go along with them.

ls

Earlier in this chapter, you learned to use the `ls` command to list the contents of a directory. The output of `ls` can be simple or extremely lengthy and complex. Both results are valuable, depending on the kind of information you need about directory contents. The `ls` command takes the syntax:

```
ls [options] [directory]
```

If you don't specify a directory, `ls` assumes that you want to know about the contents of the current working directory. You can use `ls` to get information about any directory on the system, however, as long as its permissions allow you to read the contents. (Permissions are discussed later in this chapter.)

The `ls` command offers a number of arguments that can shape the output to provide the information you need. The following table shows some of the most commonly used options. If these are not the arguments you need, consult the `ls` manual page with the command `man ls`.

Argument	Function
`-l`	Lists directory contents in long format, which shows individual file size, permissions, and other data.
`-t`	Lists directory contents sorted by the timestamp (time of the last modification).
`-a`	Lists all directory contents, including hidden files whose name begins with the `.` character.
`-i`	Lists directory contents including inode or disk index number.
`-R`	Lists directory contents including all subdirectories and their contents.

cd

The `cd` command is something that you will use frequently. It's the command used to move through the file system. It takes the syntax:

`cd` *directory*

If you issue the command without a directory destination, `cd` automatically moves you to your home directory. This is particularly useful when you've been exploring the file system and find yourself nested deep within another directory structure; just type `cd` at the command prompt and you'll return to familiar ground immediately.

Common File Manipulation Commands

After you've found the file you're looking for, there are a number of things you can do to it. This section introduces several commands used to manipulate individual files, whether they are programs, documents, or other elements treated as files by the Unix operating system.

The commands available to you depend on the Unix distribution you are using, as well as the installation choices made by your system administrator.

cat

If you want to see the contents of a given file, there's no easier way than the `cat` command, which prints the contents of a specified file to the standard output, usually the monitor. It takes the following syntax:

`cat [`*options*`]` *file(s)*

If you simply issue the command as `cat` *`filename`*, the contents print to the screen, scrolling if the file is longer than the screen length. Use a pipe to send the output through `more` or `less` (discussed in the following section) if it is too long to view in one screen. The following table shows a number of options for the `cat` command.

Argument	Function
`-n`	Numbers the output's lines
`-E`	Shows a `$` character at the end of each line
`-s`	Collapses sequential blank lines into one blank line
`-t`	Displays nonprinting tabs as `^I`
`-v`	Shows all nonprinting characters

One particularly helpful use of the `cat` command is to concatenate multiple files into one larger new file, making it easier to read the content of these files at one time. Do this with the command:

```
cat file1 file2 file3 >> newfile
```

Note the use of the redirection operator `>>` in this command.

more/less

The `more` and `less` commands are virtually identical. They are used to break up command output or file contents into single-screen chunks so that you can read the contents more easily. Both `more` and `less` can move forward through a file, although only `less` can be used to move backward. The commands take the same syntax:

```
more filename
less filename
```

When you have finished viewing the current screen of output, press the spacebar to advance to the next screen. If you are using `less` to view the output or file, you can use the B key to move back one screen.

> *If you've found the information you're looking for but you haven't scrolled through the entire file yet, just press* **Q** *in either* `more` *or* `less`*. You'll return to the command prompt.*

mv

The `mv` command is used to move a file from one location to another. It takes the syntax:

```
mv old new
```

where *`old`* is the current name of the file to be moved to the location defined as *`new`*. If the value of *`new`* is simply a new filename, the file is renamed and remains in the current directory. If the value of *`new`* is a new directory location, the file is moved to the new location with the existing filename. If the value of *`old`* is a directory, the entire directory and its contents will be moved to the location specified as *`new`*.

> *Depending on the settings on your system, if* `new` *is an existing file or directory, its contents will be overwritten by the contents of* `old`*. Be careful when reusing file and directory names.*

cp

Like the `mv` command, `cp` is used to create new files or move the content of files to another location. Unlike `mv`, however, `cp` leaves the original file intact at its original location. `cp` uses the syntax

`cp` *file1 file2*

where *`file1`* is the original file and *`file2`* is the destination file. If you use the name of an existing file as the destination value, `cp` overwrites that file's contents with the contents of *`file1`*.

rm

The `rm` command is used to delete a file. It uses the syntax:

`rm [`*options*`]` *filename*

This command can be quite destructive unless you are careful when you issue it, especially if you use wildcards. For example, the command `rm conf*` deletes all files beginning with the characters `conf`, whether you wanted to delete those files or not. The following table shows some common options for `rm`.

Argument	Function
`-i`	Forces interactive mode, prompting you to confirm each deletion
`-r`	Forces recursive mode, deleting all subdirectories and the files they contain
`-f`	Forces force mode, ignoring all warnings (very dangerous)

Be aware that combining certain options — especially the `-r` and `-f` flags — can be dangerous. The command `rm -rf *.*` would remove every single file from your file system if issued from the root directory, or every file from your home directory if issued from there. Don't do this.

touch

The `touch` command is used to update the *timestamp* on the named file. The timestamp shows the last time the file was altered or accessed. `touch` uses the syntax:

`touch` *filename*

If the filename issued as an argument does not exist, `touch` creates it as an empty file.

wc

Use the `wc` command to determine the length of a given file. `wc` uses the syntax:

`wc [`*options*`]` *filename*

By default, the output shows the length in words. The following table shows the options available for `wc`.

Argument	Function
`-c`	Shows number of individual characters (bytes) in the specified file
`-l`	Shows number of lines in the specified file
`-L`	Shows the length of the longest line in the specified file

File Ownership and Permissions

One of the distinguishing features of Unix is that it was designed from its earliest days to be a multiuser system. In contrast, it is only in recent years that other operating systems have created true multiuser functionality on a single machine. Because of its multiple-user design, Unix must use mechanisms that enable users to manage their own files without having access to the files of other users. These mechanisms are called *file ownership* and *file permissions*.

File Ownership

Any Unix user can own files. Generally, the files that the user owns are ones that he created, or which were created as a result of some action on his part. The exception to this, of course, is the *superuser*, also known as *root*. The superuser can change the ownership of any file, whether he created it or not, with the `chown` command. For example, if the superuser gives the command

```
chown jane /home/bill/billsfile
```

the ownership of the file `/home/bill/billsfile` is transferred to the user jane. Won't Bill be surprised when that happens?

Username versus UID

By now you're familiar with the idea of a username, the name you use when you log in to a Unix machine. The name is assigned to you by the system administrator (or yourself, if you're your own system administrator). In addition to a username, every user has a numerical ID number known as a user ID or UID, which is how the user is known to the system. Typically, these numbers are assigned automatically, although they can be specified when an account is created. The number itself is arbitrary, even though many systems require that ordinary users have UID numbers above 500.

The superuser always has UID 0.

For purposes other than logging in, the username and UID are basically synonymous. For example, the command

```
chown jane /home/bill/billsfile
```

could just as easily be rendered as

```
chown 503 /home/bill/billsfile
```

assuming that Jane's UID is 503. You don't need to know your UID for normal purposes, but it can come in handy.

The grep Command

The second very useful command to look at is `grep`, an unusual name that stands for *general regular expression parser*. You use `find` to search your system for files, but you use `grep` to search files for strings. Indeed, it's quite common to have `grep` as a command passed after `-exec` when using `find`.

The `grep` command takes options, a pattern to match, and files to search in:

```
grep [options] PATTERN [FILES]
```

If no filenames are given, it searches standard input.

Let's start by looking at the principal options to `grep`. Again we list only the principal options here; see the manual pages for the full list.

Option	Meaning
`-c`	Rather than print matching lines, print a count of the number of lines that match.
`-E`	Turn on extended expressions.
`-h`	Suppress the normal prefixing of each output line with the name of the file it was found in.
`-i`	Ignore case.
`-l`	List the names of the files with matching lines; don't output the actual matched line.
`-v`	Invert the matching pattern to select nonmatching lines, rather than matching lines.

Try It Out Basic grep Usage

Take a look at `grep` in action with some simple matches:

```
$ grep in words.txt
When shall we three meet again.  In thunder, lightning, or in rain?
I come, Graymalkin!
$ grep -c in words.txt words2.txt
words.txt:2
words2.txt:14
$ grep -c -v in words.txt words2.txt
words.txt:9
words2.txt:16
$
```

How It Works

The first example uses no options; it simply searches for the string "in" in the file `words.txt` and prints out any lines that match. The filename isn't printed because you are searching on just a single file.

The second example counts the number of matching lines in two different files. In this case, the filenames are printed out.

Finally, use the `-v` option to invert the search and count lines in the two files that don't match.

Regular Expressions

As you have seen, the basic usage of `grep` is very easy to master. Now it's time to look at the basics of regular expressions, which enable you to do more sophisticated matching. As mentioned earlier in the chapter, regular expressions are used in Linux and many other open-source languages. You can use them in the vi editor and in writing Perl scripts, with the basic principles common wherever they appear.

During the use of regular expressions, certain characters are processed in a special way. The most frequently used are shown in the following table:

Character	Meaning
`^`	Anchor to the beginning of a line
`$`	Anchor to the end of a line
`.`	Any single character
`[ ]`	The square braces contain a range of characters, any one of which may be matched, such as a range of characters like a–e or an inverted range by preceding the range with a ^ symbol.

If you want to use any of these characters as "normal" characters, precede them with a `\`. For example, if you wanted to look for a literal "$" character, you would simply use `\$`.

There are also some useful special match patterns that can be used in square braces, as described in the following table:

Match Pattern	Meaning
`[:alnum:]`	Alphanumeric characters
`[:alpha:]`	Letters
`[:ascii:]`	ASCII characters
`[:blank:]`	Space or tab
`[:cntrl:]`	ASCII control characters
`[:digit:]`	Digits
`[:graph:]`	Noncontrol, nonspace characters
`[:lower:]`	Lowercase letters
`[:print:]`	Printable characters
`[:punct:]`	Punctuation characters
`[:space:]`	Whitespace characters, including vertical tab
`[:upper:]`	Uppercase letters
`[:xdigit:]`	Hexadecimal digits

In addition, if the `-E` for extended matching is also specified, other characters that control the completion of matching may follow the regular expression (see the following table). With `grep` it is also necessary to precede these characters with a `\`.

Option	Meaning
`?`	Match is optional but may be matched at most once
`*`	Must be matched zero or more times
`+`	Must be matched one or more times
`{n}`	Must be matched *n* times
`{n,}`	Must be matched *n* or more times
`{n,m}`	Must be matched between *n* or *m* times, inclusive

That all looks a little complex, but if you take it in stages, you will see it's not as complex as it perhaps looks at first sight. The easiest way to get the hang of regular expressions is simply to try a few:

1. Start by looking for lines that end with the letter *e*. You can probably guess you need to use the special character `$`:

```
$ grep  e$ words2.txt
Art thou not, fatal vision, sensible
I see thee yet, in form as palpable
Nature seems dead, and wicked dreams abuse
$
```

As you can see, this finds lines that end in the letter *e*.

2. Now suppose you want to find words that end with the letter *a*. To do this, you need to use the special match characters in braces. In this case, you use `[[:blank:]]`, which tests for a space or a tab:

```
$ grep a[[:blank:]] words2.txt
Is this a dagger which I see before me,
A dagger of the mind, a false creation,
Moves like a ghost. Thou sure and firm-set earth,
$
```

3. Now look for three-letter words that start with *Th*. In this case, you need both `[[:space:]]` to delimit the end of the word and `.` to match a single additional character:

```
$ grep Th.[[:space:]] words2.txt
The handle toward my hand? Come, let me clutch thee.
The curtain'd sleep; witchcraft celebrates
Thy very stones prate of my whereabout,
$
```

4. Finally, use the extended `grep` mode to search for lowercase words that are exactly 10 characters long. Do this by specifying a range of characters to match *a* to *z*, and a repetition of 10 matches:

```
$ grep -E [a-z]\{10\} words2.txt
Proceeding from the heat-oppressed brain?
And such an instrument I was to use.
The curtain'd sleep; witchcraft celebrates
Thy very stones prate of my whereabout,
$
```

This only touches on the more important parts of regular expressions. As with most things in Linux, there is a lot more documentation out there to help you discover more details, but the best way of learning about regular expressions is to experiment.

Using gedit

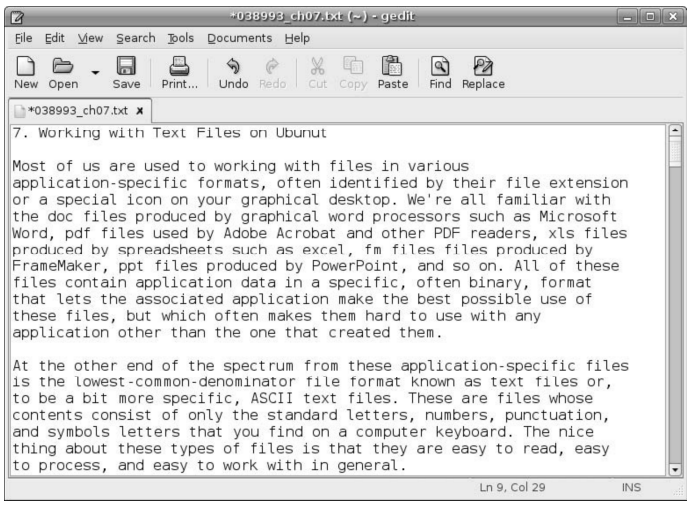
Now that I've discussed the favorite two classic Unix/Linux text editors and their attendant philosophies, it's worth mentioning that Ubuntu also provides text editors that you can use without a class in Unix history and pledging your undying allegiance to Richard Stallman or Bill Joy. Both the GNOME and KDE desktops provide easy-to-use graphical text editors called `gedit` and `kedit`, respectively. Figure 7.16 shows `gedit` displaying the same file used to illustrate the vi and emacs editors earlier in this chapter, namely, the text of this chapter as I was writing it.

The `gedit` editor is completely mouse- and menu-driven, and follows the standard keyboard conventions for most graphical editors. These include the following:

- `Control+c`: Copy selected text.
- `Control+n`: Open a new file.
- `Control+s`: Save the current file.
- `Control+v`: Paste copied or cut text.
- `Control+x`: Cut selected text.

FIGURE 7.16

The gedit text editor

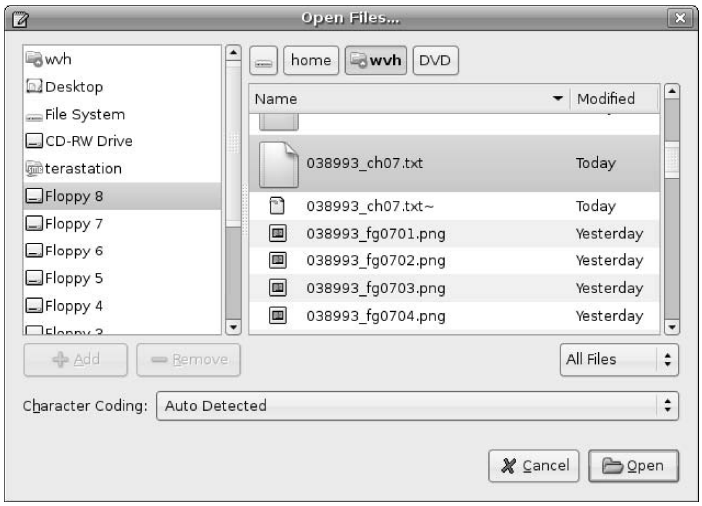


In addition to following more common keyboard conventions, `gedit` also provides convenient and more traditional dialogs for interacting with, moving around in, and selecting files and directories. Figure 7.17 shows the dialog displayed after selecting the File menu's Open command in `gedit`.

As with most GNOME applications, extensive online help for `gedit` is available by selecting the Help menu's Contents command.

FIGURE 7.17

Gedit's open file dialog



View/edit a file from the command line using vi editor

vi is a text editor which can be used directly in the terminal window using the command-line. This tutorial will cover the basics of the program, including how to create, open, edit and save documents.

Step 0: vi comes pre-installed on most Mac computers and Linux systems. If you plan to use it on a Windows system, you may need to download it and more information can be found here:

- https://www.vim.org/download.php

It may also be helpful to know the **cd** (change directory), **pwd** (print working directory) and **ls** (list contents) commands for this tutorial or to be familiar with the concept of absolute and relative paths.

Step 1: To **create a file** using vi, you can simply type ”vi” followed by the location and name of the file that you wish to create into the terminal window and press **enter**. Here is the command I use to create a file named “hello.cpp” at the location “/home/students/cking74/demo/cpp_files” of the remote machine:

```
vi /home/students/cking74/demo/cpp_files/hello.cpp
```

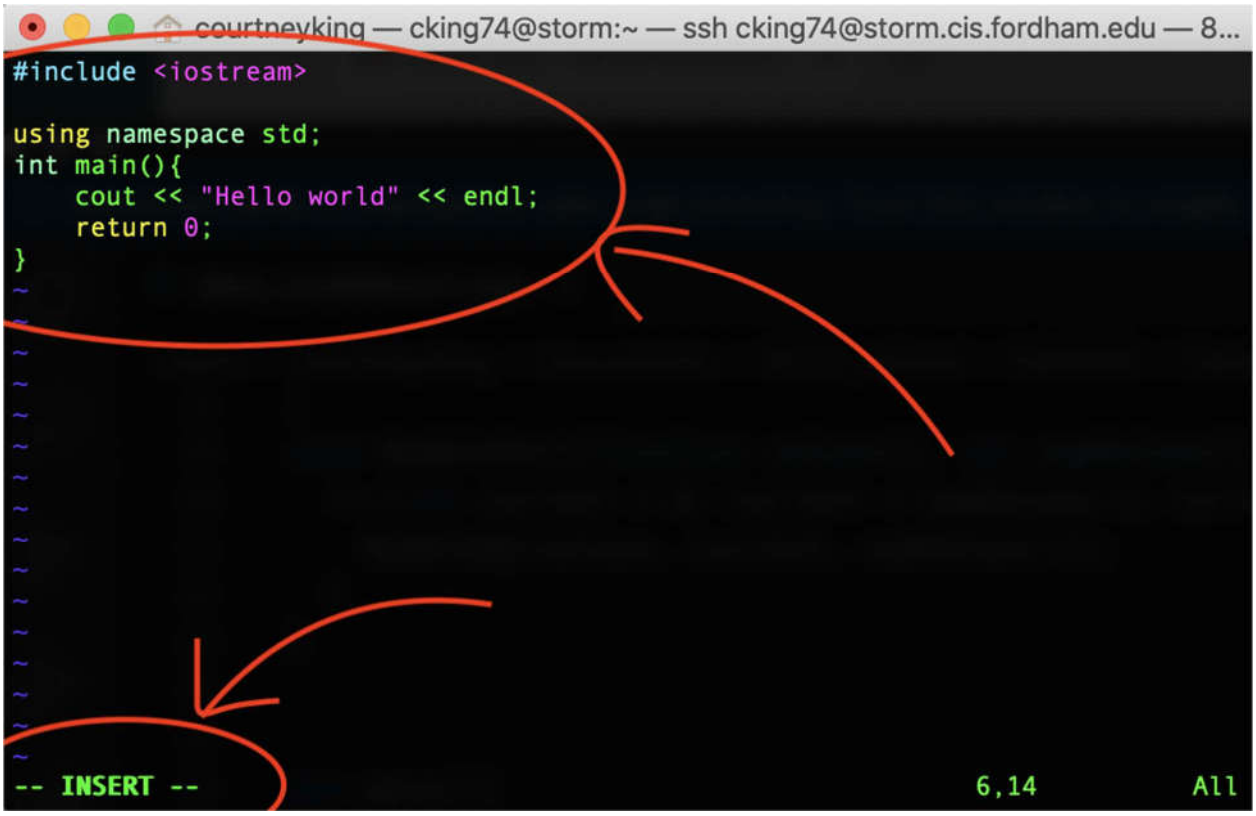
After executing the command, a blank file will appear on the screen, and you can see the location/file name and “[New]” in the bottom left corner of it as shown below. This file can be written to, saved or discarded as will be discussed in the following steps.

It is also worth noting that if any of the directories mentioned in the location used with the command do not exist, they will additionally be created in the process.

Step 2: Files can be edited using the **insert** and **visual** modes. (vi automatically opens in the **command** mode which is discussed further in step 6)

The insert mode temporarily disables the command mode and allows you to type your changes into the document using the full keyboard. To enable, press **i**. Note that pressing enter is not required to enable the mode and will result in a blank line being added to the document. While the insert mode is active – INSERT— will be displayed at the bottom of the terminal

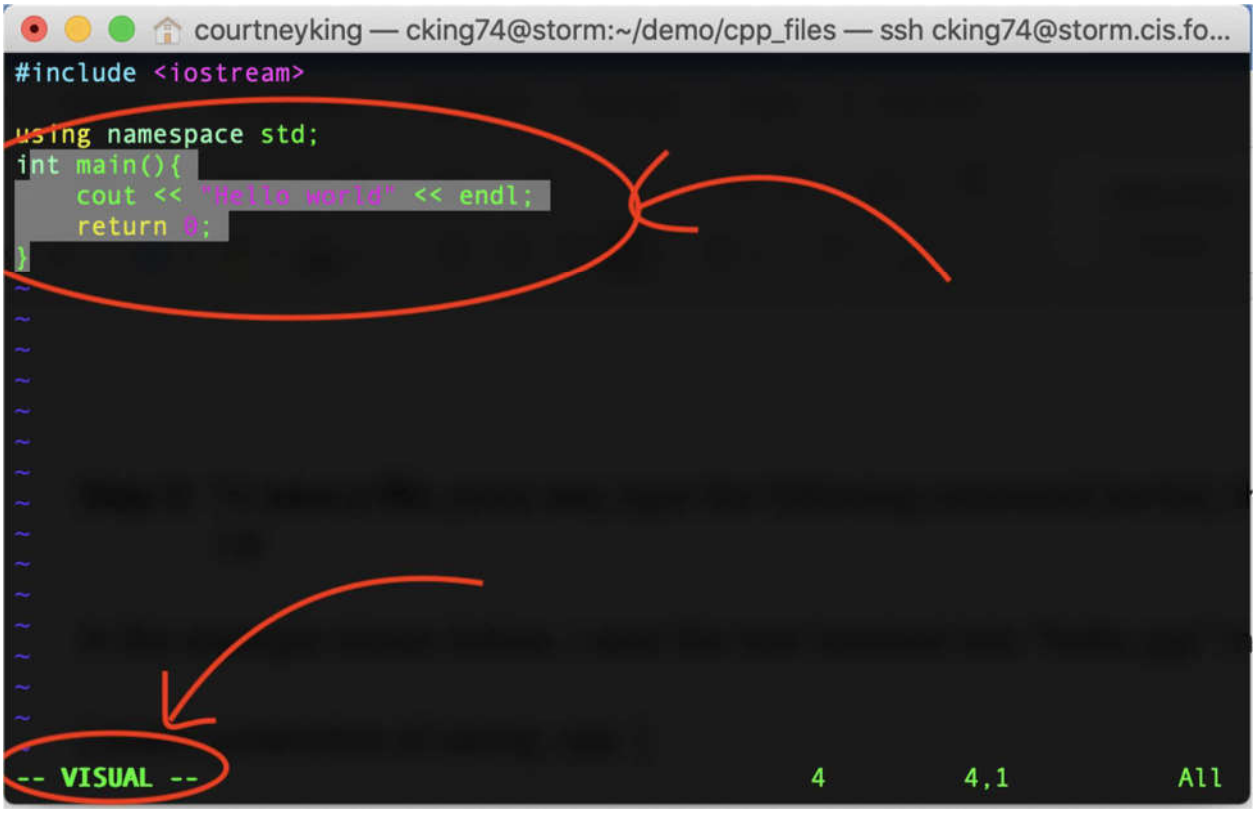
The example below shows me adding some text to the file created in the previous step using the insert mode.



The visual mode allows you to select, cut, copy and paste. It is enabled by pressing **v**. While active, –VISUAL—will be displayed at the bottom of the terminal and the following commands can be typed directly into the terminal:

v	(enable/select)
y	(copy selected text)
x	(cut selected text)
p	(paste cut/copied text)

The example below shows text being highlighted in the visual mode.

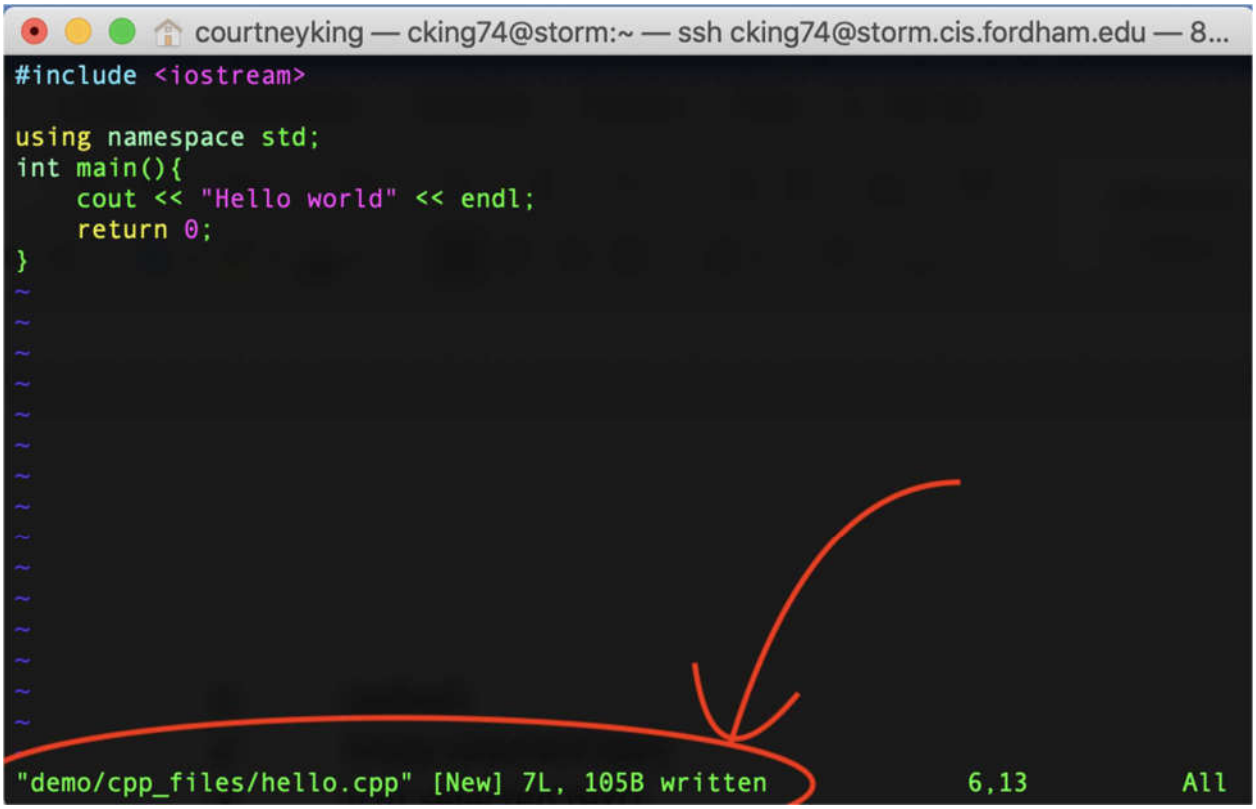


To exit either of these modes and return to command mode, press **esc**. We'll proceed to saving or discarding the changes in the following steps.

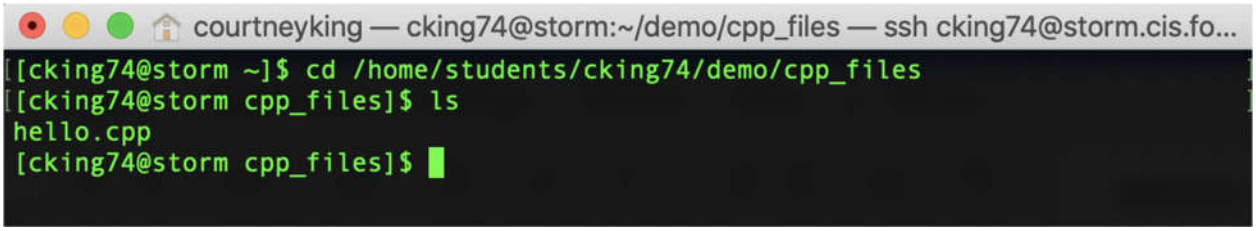
Step 3: To **save a file**, press **esc,** type the following command, then press **enter**:

:w

In the example shown below, I save the text inserted into "hello.cpp" in the previous step**.** You can see the word "written" in the bottom left corner, indicating that it has been saved at the location shown.



With a quick check, we can confirm that the file now exists in the location specified.



Step 4: To **close a file,** press **esc**, type the following command, then press **enter**:

:q

Note that if you edited the file and have unsaved changes on it that you wish to discard, you will likely get an error if you try to run the above, and need to use the following instead:

:q!

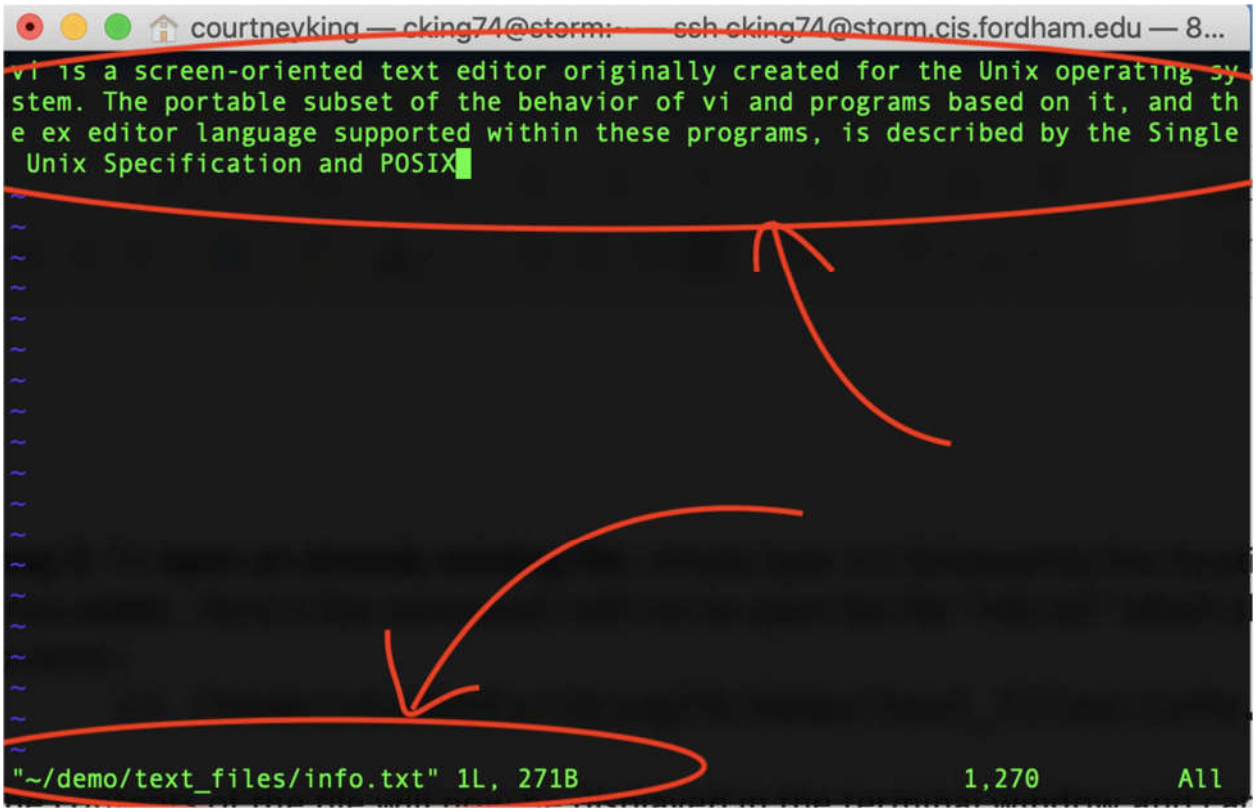
You can also combine the save and close commands into a single step with:

:wq

Step 5: To **open an already existing file**, simply type vi followed by the location and name of the file and press **enter.** Here is the command I will run to open the file "info.txt" which already exists at the specified location:

vi /home/students/cking74/demo/text_files/info.txt

The contents of the file will now be displayed in the terminal window and can be scrolled through with the arrow keys. In this example, we can see that "info.txt" contains a paragraph about vi and its location in the bottom left corner. Notice that the word [New] is not present, as was when we created a file in Step 1.



When attempting to open a file with vi, be sure to double check that the name of the file and path are correct. As shown in step 1, vi will attempt to create new files and directories for ones that don't already exist if requested, which may be confusing at first.

In the example below, I intended to open the same file "info.txt" with the following command:

vi /home/students/cking74/demo/text_files/info.txt

but as you can see, I accidentally misspelled "demo." vi attempted to create a new folder named 'dimo' when I hit **enter** (shown below)**.** Since I realized the mistake, I just closed the file without saving it (as discussed in step 4).

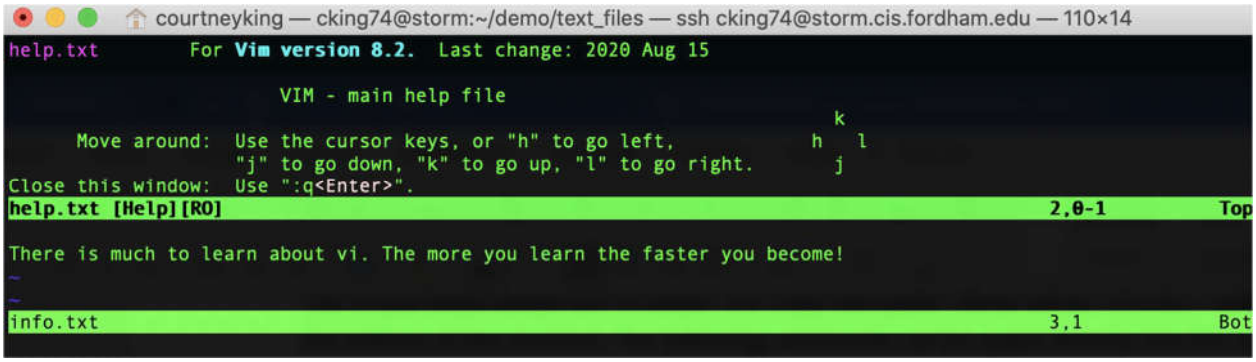
Step 6: As aforementioned, vi automatically opens in **command** mode. While in this mode, you can speed up navigating through the document by using these commands rather than arrow keys. (Try them out!):

j	(down one line)
k	(up one line)
0	(beginning of current line)
$	(end of current line)
1G	(first line of file)
nG	(nth line of file)
G	(last line of file)

Step 7: For more information and usage tips while using vi, you can type the following into terminal and press **enter:**

:help

This will display the following document, which can be read and closed by typing :q and pressing **enter.**



Congratulations! You now know the basics of vi and how to create, view and edit documents using it!

Shell Scripts: User-Defined Commands

You can place shell commands within a file and then have the shell read and execute the commands in the file. In this sense, the file functions as a shell program, executing shell commands as if they were statements in a program. A file that contains shell commands is called a *shell script*.

You enter shell commands into a script file using a standard text editor such as the Vi editor. The **sh** or **.** command used with the script's filename will read the script file and execute the commands. In the next example, the text file called **lsc** contains an **ls** command that displays only files with the extension **.c**:

```
lsc
ls *.c
```

A run of the **lsc** script is shown here:

```
$ sh lsc
main.c calc.c
$ . lsc
main.c calc.c
```

Executing Scripts

You can dispense with the **sh** and **.** commands by setting the executable permission of a script file. When the script file is first created by your text editor, it is given only read and write permission. The **chmod** command with the **+x** option will give the script file executable permission. (Permissions are discussed in Chapter 30.) Once it is executable, entering the name of the script file at the shell prompt and pressing ENTER will execute the script file and the shell commands in it. In effect, the script's filename becomes a new shell command. In this way, you can use shell scripts to design and create your own Linux commands. You need to set the permission only once. In the next example, the **lsc** file's executable permission for the owner is set to on. Then the **lsc** shell script is directly executed like any Linux command.

```
$ chmod u+x lsc
$ lsc
main.c calc.c
```

You may have to specify that the script you are using is in your current working directory. You do this by prefixing the script name with a period and slash combination, **./**, as in **./lsc**. The period is a special character representing the name of your current working directory. The slash is a directory pathname separator, as explained more fully in Chapter 32 (you could also add the current directory to your PATH variable as discussed in Chapter 9). The following example would show how you would execute the **lsc** script:

```
$ ./lsc
main.c calc.c
```

Script Arguments

Just as any Linux command can take arguments, so also can a shell script. Arguments on the command line are referenced sequentially starting with 1. An argument is referenced using the **$** operator and the number of its position. The first argument is referenced with **$1**, the second, with **$2**, and so on. In the next example, the **lsext** script prints out files with a specified extension. The first argument is the extension. The script is then executed with the argument **c** (of course, the executable permission must have been set).

lsext

```
ls *.$1
```

A run of the **lsext** script with an argument is shown here:

```
$ lsext c
main.c calc.c
```

In the next example, the commands to print out a file with line numbers have been placed in an executable file called **lpnum**, which takes a filename as its argument. The **cat** command with the **-n** option first outputs the contents of the file with line numbers. Then this output is piped into the **lpr** command, which prints it. The command to print out the line numbers is executed in the background.

lpnum

```
cat -n $1 | lpr &
```

A run of the **lpnum** script with an argument is shown here:

```
$ lpnum mydata
```

You may need to reference more than one argument at a time. The number of arguments used may vary. In **lpnum**, you may want to print out three files at one time and five files at some other time. The **$** operator with the asterisk, **$***, references all the arguments on the command line. Using **$*** enables you to create scripts that take a varying number of arguments. In the next example, **lpnum** is rewritten using **$*** so that it can take a different number of arguments each time you use it:

lpnum

```
cat -n $* | lpr &
```

A run of the **lpnum** script with multiple arguments is shown here:

```
$ lpnum mydata preface
```

Control Structures

You can control the execution of Linux commands in a shell script with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, then the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: *loops* and *conditions*. A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: **while**, **for**, and **for-in**. There are two condition structures: **if** and **case**. The control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the **if** and **while** structures, if the exit status is a zero value, then the command was successful and the structure continues.

Test Operations

With the **test** command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword **test** followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but it is written, like other options, with a minus sign and letter codes. For example, **-eq** is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality, you use the equal sign (**=**). For inequality you use **!=**. Table 8-8 lists some of the commonly used options and operators used by **test**. The syntax for the **test** command is shown here:

```
test value -option value
test string = string
```

In the next example, the user compares two integer values to see if they are equal. In this case, you need to use the equality option, **-eq**. The exit status of the **test** command is examined to find out the result of the test operation. The shell special variable **$?** holds the exit status of the most recently executed Linux command.

```
$ num=5
$ test $num -eq 10
$ echo $?
1
```

Integer Comparisons	Function
`-gt`	Greater-than
`-lt`	Less-than
`-ge`	Greater-than-or-equal-to
`-le`	Less-than-or-equal-to
`-eq`	Equal
`-ne`	Not-equal
String Comparisons	**Function**
`-z`	Tests for empty string
`=`	Equal strings
`!=`	Not-equal strings
Logical Operations	**Function**
`-a`	Logical AND
`-o`	Logical OR
`!`	Logical NOT
File Tests	**Function**
`-f`	File exists and is a regular file
`-s`	File is not empty
`-r`	File is readable
`-w`	File can be written to, modified
`-x`	File is executable
`-d`	Filename is a directory name

TABLE 8-8 BASH Shell Test Operators

Instead of using the keyword **`test`** for the **`test`** command, you can use enclosing brackets. The command **`test $greeting = "hi"`** can be written as

```
$ [ $greeting = "hi" ]
```

Similarly, the test command **`test $num -eq 10`** can be written as

```
$ [ $num -eq 10 ]
```

The brackets themselves must be surrounded by white space: a space, TAB, or ENTER. Without the spaces, it would be invalid.

Conditional Control Structures

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The **if** condition tests the success of a Linux command, not an expression. Furthermore, the end of an **if-then** command must be indicated with the keyword **fi**, and the end of a **case** command is indicated with the keyword **esac**. The condition control structures are listed in Table 8-9.

Condition Control Structures: if, else, elif, case	Function
if *command* **then** *command* **fi**	**if** executes an action if its test command is true.

if *command* **then**
command
else
command
fi
	if-else executes an action if the exit status of its test command is true; if false, then the **else** action is executed.

if *command* **then**
command
elif *command* **then**
command
else
command
fi
	elif allows you to nest **if** structures, enabling selection among several alternatives; at the first true **if** structure, its commands are executed and control leaves the entire **elif** structure.

case *string* **in** *pattern)* *command* **;;** **esac**	**case** matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed.
command && *command*	The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a nonzero value, then the AND condition is false and also returns a nonzero value.
command || *command*	The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a nonzero value, then the OR condition is false and also returns a nonzero value.
! *command*	The logical NOT condition inverts the return value of the command.

TABLE 8-9 BASH Shell Control Structures *(continued)*

Loop Control Structures: while, until, for, for-in, select	Function
while *command* **do** *command* **done**	**while** executes an action as long as its test command is true.
until *command* **do** *command* **done**	**until** executes an action as long as its test command is false.
for *variable* **in** *list-values* **do** *command* **done**	**for-in** is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.
for *variable* **do** *command* **done**	**for** is designed for reference script arguments; the variable operand is consecutively assigned each argument value.
select *string* **in** *item-list* **do** *command* **done**	**select** creates a menu based on the items in the *item-list*; then it executes the command; the command is usually a **case**.

TABLE 8-9 BASH Shell Control Structures *(continued)*

elsels

```
echo Enter s to list file sizes,
echo       otherwise all file information is listed.
echo -n "Please enter option: "
read choice
if [  "$choice" = s  ]
    then
        ls -s
    else
            ls -l
fi
echo Good-bye
```

A run of the program follows:

```
$ elsels
Enter s to list file sizes,
otherwise all file information is listed.
Please enter option: s
total 2
    1 monday     2 today
$
```

The **if** structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the **if** structure are executed. If the exit status is anything other than 0, then the command has failed and the commands within the **if** structure are not executed. The **if** command begins with the keyword **if** and is followed by a Linux command whose exit condition will be evaluated. The keyword **fi** ends the command. The **elsels** script in the next example executes the **ls** command to list files with two different possible options, either by size or with all file information. If the user enters an **s**, files are listed by size; otherwise, all file information is listed.

Loop Control Structures

The **while** loop repeats commands. A **while** loop begins with the keyword **while** and is followed by a Linux command. The keyword **do** follows on the next line. The end of the loop is specified by the keyword **done**. The Linux command used in **while** structures is often a test command indicated by enclosing brackets.

The **for-in** structure is designed to reference a list of values sequentially. It takes two operands—a variable and a list of values. The values in the list are assigned one by one to the variable in the **for-in** structure. Like the **while** command, the **for-in** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the **while** loop, the body of a **for-in** loop begins with the keyword **do** and ends with the keyword **done**. The **cbackup** script makes a backup of each file and places it in a directory called **sourcebak**. Notice the use of the ***** special character to generate a list of all filenames with a **.c** extension.

cbackup

```
for backfile in *.c
do
    cp $backfile sourcebak/$backfile
 echo $backfile
done
```

A run of the program follows:

```
$ cbackup
io.c
lib.c
main.c
$
```

The **for** structure without a specified list of values takes as its list of values the command line arguments. The arguments specified on the command line when the shell file is invoked become a list of values referenced by the **for** command. The variable used in the **for** command is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument.

The C Compiler

On POSIX-compliant systems, the C compiler is called `c89`. Historically, the C compiler was simply called `cc`. Over the years, different vendors have sold UNIX-like systems with C compilers with different facilities and options, but often still called `cc`.

When the POSIX standard was prepared, it was impossible to define a standard `cc` command with which all these vendors would be compatible. Instead, the committee decided to create a new standard command for the C compiler, `c89`. When this command is present, it will always take the same options, independent of the machine.

On Linux systems that do try to implement the standards, you might find that any or all of the commands `c89`, `cc`, and `gcc` refer to the system C compiler, usually the GNU C compiler, or `gcc`. On UNIX systems, the C compiler is almost always called `cc`.

In this book, we use `gcc` because it's provided with Linux distributions and because it supports the ANSI standard syntax for C. If you ever find yourself using a UNIX system without `gcc`, we recommend that you obtain and install it. You can find it at `http://www.gnu.org`. Wherever we use `gcc` in the book, simply substitute the relevant command on your system.

Try It Out — Your First Linux C Program

In this example you start developing for Linux using C by writing, compiling, and running your first Linux program. It might as well be that most famous of all starting points, Hello World.

1. Here's the source code for the file `hello.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("Hello World\n");
    exit(0);
}
```

2. Now compile, link, and run your program.

```
$ gcc -o hello hello.c
$ ./hello
Hello World
$
```

How It Works

You invoked the GNU C compiler (on Linux this will most likely be available as `cc` too) that translated the C source code into an executable file called `hello`. You ran the program and it printed a greeting. This is just about the simplest example there is, but if you can get this far with your system, you should be able to compile and run the remainder of the examples in the book. If this did not work for you, make sure that the C compiler is installed on your system. For example, many Linux distributions have an install option called Software Development (or something similar) that you should select to make sure the necessary packages are installed.

Because this is the first program you've run, it's a good time to point out some basics. The `hello` program will probably be in your home directory. If `PATH` doesn't include a reference to your home directory, the shell won't be able to find `hello`. Furthermore, if one of the directories in `PATH` contains another program called `hello`, that program will be executed instead. This would also happen if such a directory is mentioned in `PATH` before your home directory. To get around this potential problem, you can prefix program names with `./` (for example, `./hello`). This specifically instructs the shell to execute the program in the current directory with the given name. (The dot is an alias for the current directory.)

If you forget the `-o name` option that tells the compiler where to place the executable, the compiler will place the program in a file called `a.out` (meaning assembler output). Just remember to look for an `a.out` if you think you've compiled a program and you can't find it! In the early days of UNIX, people wanting to play games on the system often ran them as `a.out` to avoid being caught by system administrators, and some UNIX installations routinely delete all files called `a.out` every evening.

THE ONE PAGE *LINUX* MANUAL

A summary of useful Linux commands

Version 3.0 May 1999 squadron@powerup.com.au

Starting & Stopping

shutdown -h now	Shutdown the system now and do not reboot
halt	Stop all processes - same as above
shutdown -r 5	Shutdown the system in 5 minutes and reboot
shutdown -r now	Shutdown the system now and reboot
reboot	Stop all processes and then reboot - same as above
startx	Start the X system

Accessing & mounting file systems

mount -t iso9660 /dev/cdrom /mnt/cdrom	Mount the device cdrom and call it cdrom under the /mnt directory
mount -t msdos /dev/hdd /mnt/ddrive	Mount hard disk d as a msdos file system and call it ddrive under the /mnt directory
mount -t vfat /dev/hda1 /mnt/cdrive	Mount hard disk a as a VFAT file system and call it cdrive under the /mnt directory
umount /mnt/cdrom	Unmount the cdrom

Finding files and text within files

find / -name fname	Starting with the root directory, look for the file called fname
find / -name "*fname*"	Starting with the root directory, look for the file containing the string fname
locate missingfilename	Find a file called missingfilename using the locate command - this assumes you have already used the command updatedb (see next)
updatedb	Create or update the database of files on all file systems attached to the linux root directory
which missingfilename	Show the subdirectory containing the executable file called missingfilename
grep textstringtofind /dir	Starting with the directory called dir , look for and list all files containing textstringtofind

The X Window System

xvidtune	Run the X graphics tuning utility
XF86Setup	Run the X configuration menu with automatic probing of graphics cards
Xconfigurator	Run another X configuration menu with automatic probing of graphics cards
xf86config	Run a text based X configuration menu

Moving, copying, deleting & viewing files

ls -l	List files in current directory using long format
ls -F	List files in current directory and indicate the file type
ls -laC	List all files in current directory in long format and display in columns
rm name	Remove a file or directory called name
rm -rf name	Kill off an entire directory and all it s includes files and subdirectories
cp filename /home/dirname	Copy the file called filename to the /home/dirname directory
mv filename /home/dirname	Move the file called filename to the /home/dirname directory
cat filetoview	Display the file called filetoview
man -k keyword	Display man pages containing keyword
more filetoview	Display the file called filetoview one page at a time, proceed to next page using the spacebar
head filetoview	Display the first 10 lines of the file called filetoview
head -20 filetoview	Display the first 20 lines of the file called filetoview
tail filetoview	Display the last 10 lines of the file called filetoview
tail -20 filetoview	Display the last 20 lines of the file called filetoview

Installing software for Linux

rpm -ihv name.rpm	Install the rpm package called name
rpm -Uhv name.rpm	Upgrade the rpm package called name
rpm -e package	Delete the rpm package called package
rpm -l package	List the files in the package called package
rpm -ql package	List the files and state the installed version of the package called package
rpm -i --force package	Reinstall the rpm package called name having deleted parts of it (not deleting using rpm -e)
tar -zxvf archive.tar.gz or tar -zxvf archive.tgz	Decompress the files contained in the zipped and tarred archive called archive
./configure	Execute the script preparing the installed files for compiling

User Administration

adduser accountname	Create a new user call accountname
passwd accountname	Give accountname a new password
su	Log in as superuser from current login
exit	Stop being superuser and revert to normal user

Little known tips and tricks

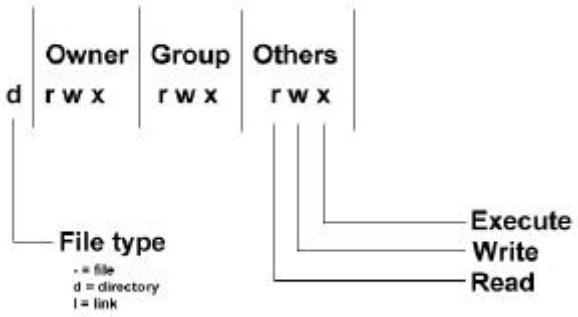
ifconfig	List ip addresses for all devices on the machine
apropos subject	List manual pages for subject
usermount	Executes graphical application for mounting and unmounting file systems

/sbin/e2fsck hda5	Execute the filesystem check utility on partition hda5
fdformat /dev/fd0H1440	Format the floppy disk in device fd0
tar -cMf /dev/fd0	Backup the contents of the current directory and subdirectories to multiple floppy disks
tail -f /var/log/messages	Display the last 10 lines of the system log.
cat /var/log/dmesg	Display the file containing the boot time messages - useful for locating problems. Alternatively, use the **dmesg** command.
*	wildcard - represents everything. eg. cp from/* to will copy all files in the from directory to the to directory
?	Single character wildcard. eg. cp config.? /configs will copy all files beginning with the name config. in the current directory to the directory named configs.
[xyz]	Choice of character wildcards. eg. ls [xyz]* will list all files in the current directory starting with the letter x, y, or z.
linux single	At the lilo prompt, start in single user mode. This is useful if you have forgotten your password. Boot in single user mode, then run the **passwd** command.
ps	List current processes
kill 123	Kill a specific process eg. kill 123

Configuration files and what they do

/etc/profile	System wide environment variables for all users.
/etc/fstab	List of devices and their associated mount points. Edit this file to add cdroms, DOS partitions and floppy drives at startup.
/etc/motd	Message of the day broadcast to all users at login.
etc/rc.d/rc.local	Bash script that is executed at the end of login process. Similar to autoexec.bat in DOS.
/etc/HOSTNAME	Conatins full hostname including domain.
/etc/cron.*	There are 4 directories that automatically execute all scripts within the directory at intervals of hour, day, week or month.
/etc/hosts	A list of all know host names and IP addresses on the machine.
/etc/httpd/conf	Paramters for the Apache web server
/etc/inittab	Specifies the run level that the machine should boot into.
/etc/resolv.conf	Defines IP addresses of DNS servers.
/etc/smb.conf	Config file for the SAMBA server. Allows file and print sharing with Microsoft clients.
~/.Xdefaults	Define configuration for some X-applications. ~ refers to user s home directory.
/etc/X11/XF86Config	Config file for X-Windows.
~/.xinitrc	Defines the windows manager loaded by X. ~ refers to user s home directory.

File permissions



If the command ls -l is given, a long list of file names is displayed. The first column in this list details the permissions applying to the file. If a permission is missing for a owner, group of other, it is represented by - eg. drwxr-x x

Read = 4 Write = 2 Execute = 1	File permissions are altered by giving the chmod command and the appropriate octal code for each user type. eg chmod 7 6 4 filename will make the file called filename R+W+X for the owner, R+W for the group and R for others.
chmod 7 5 5	Full permission for the owner, read and execute access for the group and others.
chmod +x filename	Make the file called filename executable to all users.

X Shortcuts - (mainly for Redhat)

Control|Alt + or -	Increase or decrease the screen resolution. eg. from 640x480 to 800x600
Alt | escape	Display list of active windows
Shift|Control F8	Resize the selected window
Right click on desktop background	Display menu
Shift|Control Altr	Refresh the screen
Shift|Control Altx	Start an xterm session

Printing

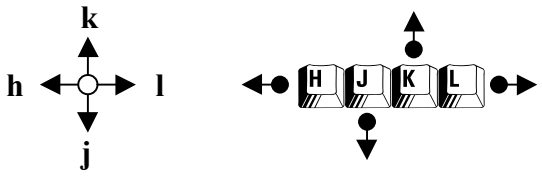
/etc/rc.d/init.d/lpd start	Start the print daemon
/etc/rc.d/init.d/lpd stop	Stop the print daemon
/etc/rc.d/init.d/lpd status	Display status of the print daemon
lpq	Display jobs in print queue
lprm	Remove jobs from queue
lpr	Print a file
lpc	Printer control tool
man subject | lpr	Print the manual page called subject as plain text
man -t subject | lpr	Print the manual page called subject as Postscript output
printtool	Start X printer setup interface

Vi Quick Reference

http://www.sfu.ca/~yzhang/linux

MOVEMENT (**lines** - ends at <CR>; **sentence** - ends at puncuation-space; **section** - ends at <EOF>)

By Character



By Line

***n*G**	to line *n*
0, $	first, last position on line
^ or _	first non-whitespace char on line
+, -	first character on next, prev line

By Screen

^F, ^B	scroll foward, back one full screen
^D, ^U	scroll forward, back half a screen
^E, ^Y	show one more line at bottom, top
L	go to the bottom of the screen
z↵	position line with cursor at top
z.	position line with cursor at middle
z-	position line with cursor at

Marking Position on Screen

m*p*	mark current position as *p* (a..z)
`*p*	move to mark position *p*
'*p*	move to first non-whitespace on line w/mark *p*

Miscellaneous Movement

f*m*	forward to character *m*
F*m*	backward to character *m*
t*m*	forward to character before *m*
T*m*	backward to character after *m*
w	move to next word (stops at puncuation)
W	move to next word (skips punctuation)
b	move to previous word (stops at punctuation)
B	move to previous word (skips punctuation)
e	end of word (puncuation not part of word)
E	end of word (punctuation part of word)
), (	next, previous sentence
]], [[	next, previous section
}, {	next, previous paragraph
%	goto matching parenthesis () {} []

EDITING TEXT

Entering Text

a	append after cursor
A or $a	append at end of line
i	insert before cursor
I or _i	insert at beginning of line
o	open line below cursor
O	open line above cursor
c*m*	change text (*m* is movement)

Cut, Copy, Paste (Working w/Buffers)

d*m*	delete (*m* is movement)
dd	delete line
D or d$	delete to end of line
x	delete char under cursor
X	delete char before cursor
y*m*	yank to buffer (*m* is movement)
yy or Y	yank to buffer current line
p	paste from buffer after cursor
P	paste from buffer before cursor
"*b*dd	cut line into named buffer *b* (a..z)
"*b*p	paste from named buffer *b*

Searching and Replacing

/*w*	search forward for *w*
?*w*	search backward for *w*
/*w*/+*n*	search forward for *w* and move down *n* lines
n	repeat search (forward)
N	repeat search (backward)

:s/*old*/*new*	replace next occurence of *old* with *new*
:s/*old*/*new*/g	replace all occurences on the line
:*x*,*y*s/*old*/*new*/g	replace all ocurrences from line *x* to *y*
:%s/*old*/*new*/g	replace all occurrences in file
:%s/*old*/*new*/gc	same as above, with confirmation

Miscellaneous

n*>*m	indent *n* lines (*m* is movement)
n*<*m	un-indent left *n* lines (*m* is movement)
.	repeat last command
U	undo changes on current line
u	undo last command
J	join end of line with next line (at <cr>)
:r*f*	insert text from external file *f*
^G	show status