Introduction

I want to tell you a story.

No, not the story of how, in 1991, Linus Torvalds wrote the first version of the Linux kernel. You can read that story in lots of Linux books. Nor am I going to tell you the story of how, some years earlier, Richard Stallman began the GNU Project to create a free Unix-like operating system. That's an important story too, but most other Linux books have that one, as well.

No, I want to tell you the story of how you can take back control of your computer.

When I began working with computers as a college student in the late 1970s, there was a revolution going on. The invention of the microprocessor had made it possible for ordinary people like you and me to actually own a computer. It's hard for many people today to imagine what the world was like when only big business and big government ran all the computers. Let's just say, you couldn't get much done.

Today, the world is very different. Computers are everywhere, from tiny wristwatches to giant data centers to everything in between. In addition to ubiquitous computers, we also have a ubiquitous network connecting them together. This has created a wondrous new age of personal empowerment and creative freedom, but over the last couple of decades something else has been happening. A few giant corporations have been imposing their control over most of the world's computers and deciding what you can and cannot do with them. Fortunately, people from all over the world are doing something about it. They are fighting to maintain control of their computers by writing their own software. They are building Linux.

Many people speak of "freedom" with regard to Linux, but I don't think most people know what this freedom really means. Freedom is the power to decide what your computer does, and the only way to have this freedom is to know what your computer is doing. Freedom is a computer that is without secrets, one where everything can be known if you care enough to find out.

Why Use The Command Line?

Have you ever noticed in the movies when the "super hacker,"—you know, the guy who can break into the ultra-secure military computer in under thirty seconds—sits down at the computer, he never touches a mouse? It's because movie makers realize that we, as human beings, instinctively know the only way to really get anything done on a computer leges.

Assuming that things are good so far, let's try some typing. Enter some gibberish at the prompt like so:

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

Since this command makes no sense, the shell will tell us so and give us another chance:

```
bash: kaekfjaeifj: command not found
[me@linuxbox ~]$
```

Command History

If we press the up-arrow key, we will see that the previous command "kaekfjaeifj" reappears after the prompt. This is called *command history*. Most Linux distributions remember the last 500 commands by default. Press the down-arrow key and the previous command disappears.

Cursor Movement

Recall the previous command with the up-arrow key again. Now try the left and right-arrow keys. See how we can position the cursor anywhere on the command line? This makes editing commands easy.

A Few Words About Mice And Focus

While the shell is all about the keyboard, you can also use a mouse with your terminal emulator. There is a mechanism built into the X Window System (the underlying engine that makes the GUI go) that supports a quick copy and paste technique. If you highlight some text by holding down the left mouse button and dragging the mouse over it (or double clicking on a word), it is copied into a buffer maintained by X. Pressing the middle mouse button will cause the text to be pasted at the cursor location. Try it.

Note: Don't be tempted to use `Ctrl-c` and `Ctrl-v` to perform copy and paste inside a terminal window. They don't work. These control codes have different meanings to the shell and were assigned many years before Microsoft Windows.

2 – Navigation

The first thing we need to learn (besides just typing) is how to navigate the file system on our Linux system. In this chapter we will introduce the following commands:

- `pwd` - Print name of current working directory
- `cd` - Change directory
- `ls` - List directory contents

Understanding The File System Tree

Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a *hierarchical directory structure*. This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories. The first directory in the file system is called the *root directory*. The root directory contains files and subdirectories, which contain more files and subdirectories and so on and so on.

Note that unlike Windows, which has a separate file system tree for each storage device, Unix-like systems such as Linux always have a single file system tree, regardless of how many drives or storage devices are attached to the computer. Storage devices are attached (or more correctly, *mounted*) at various points on the tree according to the whims of the *system administrator*, the person (or persons) responsible for the maintenance of the system.

The Current Working Directory

Most of us are probably familiar with a graphical file manager which represents the file system tree as in Figure 1. Notice that the tree is usually shown upended, that is, with the root at the top and the various branches descending below.

However, the command line has no pictures, so to navigate the file system tree we need to think of it in a different way.

`Data???`	Any file beginning with "Data" followed by exactly three characters
`[abc]*`	Any file beginning with either an "a", a "b", or a "c"
`BACKUP.[0-9][0-9][0-9]`	Any file beginning with "BACKUP." followed by exactly three numerals
`[[:upper:]]*`	Any file beginning with an uppercase letter
`[![:digit:]]*`	Any file not beginning with a numeral
`*[[:lower:]123]`	Any file ending with a lowercase letter or the numerals "1", "2", or "3"

Wildcards can be used with any command that accepts filenames as arguments, but we'll talk more about that in Chapter 7.

Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` or the `[a-z]` character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be very careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

Wildcards Work In The GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line, but are also supported by some graphical file managers.

- In **Nautilus** (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.
- In some versions of **Dolphin** and **Konqueror** (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase "u" in the /usr/bin directory, enter "/usr/bin/u*" in the location bar and it will display the result.

Great! “foo” is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command:

```
alias name='string'
```

After the command “alias” we give alias a name followed immediately (no whitespace allowed) by an equals sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, it can be used anywhere the shell would expect a command. Let's try it:

```
[me@linuxbox ~]$ foo
bin   games    kerberos  lib64    local  share  tmp
etc   include  lib       libexec  sbin   src
/home/me
[me@linuxbox ~]$
```


We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls ; cd -'
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output and error, the shell references them internally as file descriptors 0, 1 and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor "2" is placed immediately before the redirection operator to perform the redirection of standard error to the file `ls-error.txt`.

Redirecting Standard Output And Standard Error To One File

There are cases in which we may wish to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. First, the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```


Using this method, we perform two redirections. First we redirect standard output to the file `ls-output.txt` and then we redirect file descriptor 2 (standard error) to file descriptor one (standard output) using the notation `2>&1`.

Notice that the order of the redirections is significant. The redirection of standard error must always occur *after* redirecting standard output or it doesn't work. In the example above,

```
>ls-output.txt 2>&1
```

redirects standard error to the file `ls-output.txt`, but if the order is changed to

```
 2>&1 >ls-output.txt
```

standard error is directed to the screen.

Recent versions of `bash` provide a second, more streamlined method for performing this

```
cat [file...]
```

In most cases, you can think of `cat` as being analogous to the `TYPE` command in DOS. You can use it to display files without paging, for example:

```
[me@linuxbox ~]$ cat ls-output.txt
```

will display the contents of the file `ls-output.txt`. `cat` is often used to display short text files. Since `cat` can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named:

`movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099`

we could join them back together with this command:

```
cat movie.mpeg.0* > movie.mpeg
```


Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter “cat” with no arguments:

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

Nothing happens, it just sits there like it's hung. It may seem that way, but it's really doing exactly what it's supposed to.

If `cat` is not given any arguments, it reads from standard input and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try adding the following text and pressing Enter:

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

Next, type a `Ctrl-d` (i.e., hold down the Ctrl key and press “d”) to tell `cat` that it has utilized by a shell feature called *pipelines*. Using the pipe operator "|" (vertical bar), the standard output of one command can be *piped* into the standard input of another:

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's `less`. We can use `less` to display, page-by-page, the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

The Difference Between > and |

At first glance, it may be hard to understand the redirection performed by the pipeline operator | versus the redirection operator >. Simply put, the redirection operator connects a command with a file while the pipeline operator connects the output of one command with the input of a second command.

```
command1 > file1
command1 | command2
```

A lot of people will try the following when they are learning about pipelines, "just to see what happens."

```
command1 > command2
```

Answer: Sometimes something really bad.

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

```
# cd /usr/bin
# ls > less
```

7 – Seeing The World As The Shell Sees It

In this chapter we are going to look at some of the "magic" that occurs on the command line when you press the enter key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

- `echo` – Display a line of text

Expansion

Each time you type a command line and press the enter key, `bash` performs several processes upon the text before it carries out your command. We have seen a couple of cases of how a simple character sequence, for example "*", can have a lot of meaning to the shell. The process that makes this happen is called *expansion*. With expansion, you enter something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the `echo` command. `echo` is a shell builtin that performs a very simple task. It prints out its text arguments on standard output.


```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to `echo` gets displayed. Let's try another example:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

So what just happened? Why didn't `echo` print "*"? As you recall from our work with wildcards, the "*" character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the "*" into something else (in this instance, the names of the files in the

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Here is an example using the division and remainder operators. Notice the effect of integer division:

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

Arithmetic expansion is covered in greater detail in Chapter 34.

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, you can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```


Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

Integers may also be *zero-padded* like so:

```
[me@linuxbox ~]$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
[me@linuxbox ~]$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
```

A range of letters in reverse order:

Table 8-3: Cut And Paste Commands

Key	Action
`Ctrl-k`	Kill text from the cursor location to the end of line.
`Ctrl-u`	Kill text from the cursor location to the beginning of the line.
`Alt-d`	Kill text from the cursor location to the end of the current word.
`Alt-Backspace`	Kill text from the cursor location to the beginning of the current word. If the cursor is at the beginning of a word, kill the previous word.
`Ctrl-y`	Yank text from the kill-ring and insert it at the cursor location.

The Meta Key

If you venture into the Readline documentation, which can be found in the READLINE section of the `bash` man page, you will encounter the term "meta key." On modern keyboards this maps to the `Alt` key but it wasn't always so.

Back in the dim times (before PCs but after Unix) not everybody had their own computer. What they might have had was a device called a *terminal*. A terminal was a communication device that featured a text display screen and a keyboard and just enough electronics inside to display text characters and move the cursor around. It was attached (usually by serial cable) to a larger computer or the communication network of a larger computer. There were many different brands of terminals and they all had different keyboards and display feature sets. Since they all tended to at least understand ASCII, software developers wanting portable applications wrote to the lowest common denominator. Unix systems have a very elaborate way of dealing with terminals and their different display features. Since the developers of Readline could not be sure of the presence of a dedicated extra control key, they invented one and called it "meta." While the `Alt` key serves as the meta key on modern keyboards, you can also press and release the `Esc` key to get the same effect as holding down the `Alt` key if you're still using a terminal (which you can still do in Linux!).

Completion

Another way that the shell can help you is through a mechanism called *completion*. Completion occurs when you press the tab key while typing a command. Let's see how this works. Given a home directory that looks like this:

completion will also work on variables (if the beginning of the word is a "$"), user names (if the word begins with "~"), commands (if the word is the first word on the line.) and hostnames (if the beginning of the word is "@"). Hostname completion only works for hostnames listed in `/etc/hosts`.

There are a number of control and meta key sequences that are associated with completion:

Table 8-4: Completion Commands

Key	Action
`Alt-?`	Display list of possible completions. *On most systems you can also do this by pressing the tab key a second time, which is much easier.*
`Alt-*`	Insert all possible completions. This is useful when you want to use more than one possible match.

There quite a few more that I find rather obscure. You can see a list in the `bash` man page under "READLINE".

Programmable Completion

Recent versions of `bash` have a facility called *programmable completion*. Programmable completion allows you (or more likely, your distribution provider) to add additional completion rules. Usually this is done to add support for specific applications. For example it is possible to add completions for the option list of a command or match particular file types that an application supports. Ubuntu has a fairly large set defined by default. Programmable completion is implemented by shell functions, a kind of mini shell script that we will cover in later chapters. If you are curious, try:

```
set | less
```

and see if you can find them. Not all distributions include them by default.

Using History

As we discovered in Chapter 1, `bash` maintains a history of commands that have been entered. This list of commands is kept in your home directory in a file called `.bash_history`. The history facility is a useful resource for reducing the amount of typing you have to do, especially when combined with command line editing.

If no character is specified, “all” will be assumed. The operation may be a “+” indicating that a permission is to be added, a “-” indicating that a permission is to be taken away, or a “=” indicating that only the specified permissions are to be applied and that all others are to be removed.

Permissions are specified with the “r”, “w”, and “x” characters. Here are some examples of symbolic notation:

Table 9-6: chmod Symbolic Notation Examples

Notation	Meaning
`u+x`	Add execute permission for the owner.
`u-x`	Remove execute permission from the owner.
`+x`	Add execute permission for the owner, group, and world. Equivalent to `a+x`.
`o-rw`	Remove the read and write permission from anyone besides the owner and group owner.
`go=rw`	Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or world previously had execute permissions, they are removed.
`u+x,go=rx`	Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

Some people prefer to use octal notation, some folks really like the symbolic. Symbolic notation does offer the advantage of allowing you to set a single attribute without disturbing any of the others.

Take a look at the `chmod` man page for more details and a list of options. A word of caution regarding the “--recursive” option: it acts on both files and directories, so it's not as useful as one would hope since, we rarely want files and directories to have the same permissions.

Setting File Mode With The GUI

Now that we have seen how the permissions on files and directories are set, we can better understand the permission dialogs in the GUI. In both Nautilus (GNOME) and Konqueror (KDE), right-clicking a file or directory icon will expose a properties dialog. Here is an example from KDE 3.5:

with the value `0002` (the value `0022` is another common default value), which is the octal representation of our mask. We next create a new instance of the file `foo.txt` and observe its permissions.

We can see that both the owner and group get read and write permission, while everyone else only gets read permission. The reason that world does not have write permission is because of the value of the mask. Let's repeat our example, this time setting the mask ourselves:

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me   0 2008-03-06 14:58 foo.txt
```

When we set the mask to `0000` (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we take the mask and expand it into binary, and then compare it to the attributes we can see what happens:

Original file mode	`--- rw- rw- rw-`
Mask	`000 000 000 010`
Result	`--- rw- rw- r--`

Ignore for the moment the leading zeros (we'll get to those in a minute) and observe that where the 1 appears in our mask, an attribute was removed—in this case, the world write permission. That's what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of `0022`, we can see what it does:

Original file mode	`--- rw- rw- rw-`
Mask	`000 000 010 010`
Result	`--- rw- r-- r--`

Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some sevens) to get used to how this works. When you're done, remember to clean up:

changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the "-l" may be abbreviated "-", which is how it is most often used. To start a shell for the superuser, we would do this:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]#
```

After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges (the trailing "#" rather than a "$") and the current working directory is now the home directory for the superuser (normally `/root`.) Once in the new shell, we can carry out commands as the superuser. When finished, enter "exit" to return to the previous shell:

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

It is also possible to execute a single command rather than starting a new interactive command by using `su` this way:


```
su -c 'command'
```

Using this form, a single command line is passed to the new shell for execution. It is important to enclose the command in quotes, as we do not want expansion to occur in our shell, but rather in the new shell:

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Password:
-rw------- 1 root root     754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
total 0
[me@linuxbox ~]$
```

`:admins`	Changes the group owner to the group `admins`. The file owner is unchanged.
`bob:`	Change the file owner from the current owner to user `bob` and changes the group owner to the login group of user `bob`.

Let's say that we have two users; `janet`, who has access to superuser privileges and `tony`, who does not. User `janet` wants to copy a file from her home directory to the home directory of user `tony`. Since user `janet` wants `tony` to be able to edit the file, `janet` changes the ownership of the copied file from `janet` to `tony`:

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root  root  8031 2008-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony  tony  8031 2008-03-20 14:30 /home/tony/myfile.txt
```

Here we see user `janet` copy the file from her directory to the home directory of user `tony`. Next, `janet` changes the ownership of the file from `root` (a result of using `sudo`) to `tony`. Using the trailing colon in the first argument, `janet` also changed the group ownership of the file to the login group of `tony`, which happens to be group `tony`.

Notice that after the first use of `sudo`, `janet` was not prompted for her password? This is because `sudo`, in most configurations, "trusts" you for several minutes until its timer runs out.

chgrp – Change Group Ownership

In older versions of Unix, the `chown` command only changed file ownership, not group ownership. For that purpose, a separate command, `chgrp` was used. It works much the same way as `chown`, except for being more limited.

Exercising Our Privileges

Now that we have learned how this permissions thing works, it's time to show it off. We are going to demonstrate the solution to a common problem—setting up a shared directory. Let's imagine that we have two users named "bill" and "karen." They both have music CD collections and wish to set up a shared directory, where they will each store their

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2008-03-21 18:05 /usr/local/share/Music
```

So what does this all mean? It means that we now have a directory, `/usr/local/share/Music` that is owned by `root` and allows read and write access to group `music`. Group `music` has members `bill` and `karen`, thus `bill` and `karen` can create files in directory `/usr/local/share/Music`. Other users can list the contents of the directory but cannot create files there.

But we still have a problem. With the current permissions, files and directories created within the Music directory will have the normal permissions of the users `bill` and `karen`:

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill   bill   0 2008-03-24 20:03 test_file
```


Actually there are two problems. First, the default `umask` on this system is 0022 which prevents group members from writing files belonging to other members of the group. This would not be a problem if the shared directory only contained files, but since this directory will store music, and music is usually organized in a hierarchy of artists and albums, members of the group will need the ability to create files and directories inside directories created by other members. We need to change the `umask` used by `bill` and `karen` to 0002 instead.

Second, each file and directory created by one member will be set to the primary group of the user rather than the group `music`. This can be fixed by setting the setgid bit on the directory:

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2008-03-24 20:03 /usr/local/share/Music
```

Now we test to see if the new permissions fix the problem. `bill` sets his `umask` to 0002, removes the previous test file, and creates a new test file and directory:

```
[bill@linuxbox ~]$ umask 0002
```

```
[me@linuxbox ~]$ ps aux
USER       PID %CPU %MEM    VSZ    RSS TTY       STAT START   TIME COMMAND
root         1  0.0  0.0   2136    644 ?         Ss   Mar05   0:31 init
root         2  0.0  0.0      0      0 ?         S<   Mar05   0:00 [kt]
root         3  0.0  0.0      0      0 ?         S<   Mar05   0:00 [mi]
root         4  0.0  0.0      0      0 ?         S<   Mar05   0:00 [ks]
root         5  0.0  0.0      0      0 ?         S<   Mar05   0:06 [wa]
root         6  0.0  0.0      0      0 ?         S<   Mar05   0:36 [ev]
root         7  0.0  0.0      0      0 ?         S<   Mar05   0:00 [kh]


and many more...
```

This set of options displays the processes belonging to every user. Using the options without the leading dash invokes the command with "BSD style" behavior. The Linux version of `ps` can emulate the behavior of the `ps` program found in several different Unix implementations. With these options, we get these additional columns:

Table 10-2: BSD Style ps Column Headers

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage in percent.
%MEM	Memory usage in percent.
VSZ	Virtual memory size.
RSS	Resident Set Size. The amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over 24 hours, a date is used.

Viewing Processes Dynamically With `top`

While the `ps` command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the `ps` command is executed. To see a more dynamic view of the machine's activity, we use the `top` command:

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

experiments, we're going to use a little program called `xlogo` as our guinea pig. The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go) which simply displays a re-sizable window containing the X logo. First, we'll get to know our test subject:

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

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` may print a warning message, but it may be safely ignored.

Tip: If your system does not include the `xlogo` program, try using `gedit` or `kwrite` instead.

We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the `xlogo` window, the prompt returns.

Interrupting A Process

Let's observe what happens when we run `xlogo` again. First, enter the `xlogo` command and verify that the program is running. Next, return to the terminal window and press `Ctrl-c`.

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

In a terminal, pressing `Ctrl-c`, *interrupts* a program. This means that we politely asked the program to terminate. After we pressed `Ctrl-c`, the `xlogo` window closed and the shell prompt returned.

Many (but not all) command-line programs can be interrupted by using this technique.

Putting A Process In The Background

Let's say we wanted to get the shell prompt back without terminating the `xlogo` pro-

		the program but the program may choose to ignore it.
28	WINCH	Window Change. This is a signal sent by the system when a window changes size. Some programs , like `top` and `less` will respond to this signal by redrawing themselves to fit the new window dimensions.

For the curious, a complete list of signals can be seen with the following command:

```
[me@linuxbox ~]$ kill -l
```

Sending Signals To Multiple Processes With `killall`

It's also possible to send signals to multiple processes matching a specified program or username by using the `killall` command. Here is the syntax:

```
killall [-u user] [-signal] name...
```


To demonstrate, we will start a couple of instances of the `xlogo` program and then terminate them:

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]-  Terminated              xlogo
[2]+  Terminated              xlogo
```

Remember, as with `kill`, you must have superuser privileges to send signals to processes that do not belong to you.

More Process Related Commands

Since monitoring processes is an important system administration task, there are a lot of commands for it. Here are some to play with:

Table 10-6: Other Process Related Commands

Command	Description
`pstree`	Outputs a process list arranged in a tree-like pattern showing the parent/child relationships between processes.
`vmstat`	Outputs a snapshot of system resource usage including, memory, swap and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. For example: `vmstat 5`. Terminate the output with `Ctrl-c`.
`xload`	A graphical program that draws a graph showing system load over time.
`tload`	Similar to the `xload` program, but draws the graph in the terminal. Terminate the output with `Ctrl-c`.

Summing Up

Most modern systems feature a mechanism for managing multiple processes. Linux provides a rich set of tools for this purpose. Given that Linux is the world's most deployed server operating system, this makes a lot of sense. However, unlike some other systems, Linux relies primarily on command line tools for process management. Though there are graphical process tools for Linux, the command line tools are greatly preferred because of their speed and light footprint. While the GUI tools may look pretty, they often create a lot of system load themselves, which somewhat defeats the purpose.

them, and since programmers use them extensively, they write editors to express their own desires as to how they should work.

Text editors fall into two basic categories: graphical and text based. GNOME and KDE both include some popular graphical editors. GNOME ships with an editor called `gedit`, which is usually called "Text Editor" in the GNOME menu. KDE usually ships with three which are (in order of increasing complexity) `kedit`, `kwrite`, and `kate`.

There are many text-based editors. The popular ones you will encounter are `nano`, `vi`, and `emacs`. The `nano` editor is a simple, easy-to-use editor designed as a replacement for the `pico` editor supplied with the PINE email suite. The `vi` editor (on most Linux systems replaced by a program named `vim`, which is short for "Vi IMproved") is the traditional editor for Unix-like systems. It will be the subject of our next chapter. The `emacs` editor was originally written by Richard Stallman. It is a gigantic, all-purpose, does-everything programming environment. While readily available, it is seldom installed on most Linux systems by default.

Using A Text Editor

All text editors can be invoked from the command line by typing the name of the editor followed by the name of the file you want to edit. If the file does not already exist, the editor will assume that you want to create a new file. Here is an example using `gedit`:


```
[me@linuxbox ~]$ gedit some_file
```

This command will start the `gedit` text editor and load the file named "some_file", if it exists.

All graphical text editors are pretty self-explanatory, so we won't cover them here. Instead, we will concentrate on our first text-based text editor, `nano`. Let's fire up `nano` and edit the `.bashrc` file. But before we do that, let's practice some "safe computing." Whenever we edit an important configuration file, it is always a good idea to create a backup copy of the file first. This protects us in case we mess the file up while editing. To create a backup of the `.bashrc` file, do this:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

It doesn't matter what you call the backup file, just pick an understandable name. The extensions ".bak", ".sav", ".old", and ".orig" are all popular ways of indicating a backup file. Oh, and remember that `cp` will *overwrite existing files* silently.

A Little Background

The first version of `vi` was written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems. `vi` derives its name from the word “visual,” because it was intended to allow editing on a video terminal with a moving cursor. Previous to *visual editors*, there were *line editors* which operated on a single line of text at a time. To specify a change, we tell a line editor to go to a particular line and describe what change to make, such as adding or deleting text. With the advent of video terminals (rather than printer-based terminals like teletypes) visual editing became possible. `vi` actually incorporates a powerful line editor called `ex`, and we can use line editing commands while using `vi`.

Most Linux distributions don't include real `vi`; rather, they ship with an enhanced replacement called `vim` (which is short for “vi improved”) written by Bram Moolenaar. `vim` is a substantial improvement over traditional Unix `vi` and is usually symbolically linked (or aliased) to the name “vi” on Linux systems. In the discussions that follow, we will assume that we have a program called “vi” that is really `vim`.

Starting And Stopping vi

To start `vi`, we simply enter the following:


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

And a screen like this should appear:

```
~
~
~                          VIM - Vi Improved
~
~                           version 7.1.138
~                      by Bram Moolenaar et al.
~             Vim is open source and freely distributable
~
~                      Sponsor Vim development!
~           type  :help sponsor<Enter>    for information
~
~           type  :q<Enter>               to exit
~           type  :help<Enter>  or  <F1>  for on-line help
~           type  :help version7<Enter>   for version info
~
~                    Running in Vi compatible mode
~           type  :set nocp<Enter>        for Vim defaults
```

nally written, not all video terminals had arrow keys, and skilled typists could use regular keyboard keys to move the cursor without ever having to lift their fingers from the keyboard.

Many commands in `vi` can be prefixed with a number, as with the “G” command listed above. By prefixing a command with a number, we may specify the number of times a command is to be carried out. For example, the command “5j” causes `vi` to move the cursor down five lines.

Basic Editing

Most editing consists of a few basic operations such as inserting text, deleting text, and moving text around by cutting and pasting. `vi`, of course, supports all of these operations in its own unique way. `vi` also provides a limited form of undo. If we press the “u” key while in command mode, `vi` will undo the last change that you made. This will come in handy as we try out some of the basic editing commands.

Appending Text

`vi` has several different ways of entering insert mode. We have already used the `i` command to insert text.

Let's go back to our `foo.txt` file for a moment:


```
The quick brown fox jumped over the lazy dog.
```

If we wanted to add some text to the end of this sentence, we would discover that the `i` command will not do it, since we can't move the cursor beyond the end of the line. `vi` provides a command to append text, the sensibly named “a” command. If we move the cursor to the end of the line and type “a”, the cursor will move past the end of the line and `vi` will enter insert mode. This will allow us to add some more text:

```
The quick brown fox jumped over the lazy dog. It was cool.
```

Remember to press the `Esc` key to exit insert mode.

Since we will almost always want to append text to the end of a line, `vi` offers a shortcut to move to the end of the current line and start appending. It's the “A” command. Let's try it and add some more lines to our file.

First, we'll move the cursor to the beginning of the line using the “0” (zero) command.

```
Line 3
Line 4
Line 5
```

Exit insert mode by pressing the `Esc` key and undo our change by pressing `u`.

Deleting Text

As we might expect, `vi` offers a variety of ways to delete text, all of which contain one of two keystrokes. First, the `x` key will delete a character at the cursor location. `x` may be preceded by a number specifying how many characters are to be deleted. The `d` key is more general purpose. Like `x`, it may be preceded by a number specifying the number of times the deletion is to be performed. In addition, `d` is always followed by a movement command that controls the size of the deletion. Here are some examples:

Table 12-3: Text Deletion Commands

Command	Deletes
`x`	The current character.
`3x`	The current character and the next two characters.
`dd`	The current line.
`5dd`	The current line and the next four lines.
`dW`	From the current cursor position to the beginning of the next word.
`d$`	From the current cursor location to the end of the current line.
`d0`	From the current cursor location to the beginning of the line.
`d^`	From the current cursor location to the first non-whitespace character in the line.
`dG`	From the current line to the end of the file.
`d20G`	From the current line to the twentieth line of the file.

Place the cursor on the word “It” on the first line of our text. Press the `x` key repeatedly until the rest of the sentence is deleted. Next, press the `u` key repeatedly until the deletion used to cut text. Here are some examples combining the `y` command with various movement commands:

Table13- 4: Yanking Commands

Command	Copies
`yy`	The current line.
`5yy`	The current line and the next four lines.
`yW`	From the current cursor position to the beginning of the next word.
`y$`	From the current cursor location to the end of the current line.
`y0`	From the current cursor location to the beginning of the line.
`y^`	From the current cursor location to the first non-whitespace character in the line.
`yG`	From the current line to the end of the file.
`y20G`	From the current line to the twentieth line of the file.

Let's try some copy and paste. Place the cursor on the first line of the text and type `yy` to copy the current line. Next, move the cursor to the last line (`G`) and type `p` to paste the line below the current line:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
The quick brown fox jumped over the lazy dog. It was cool.
```

Just as before, the `u` command will undo our change. With the cursor still positioned on the last line of the file, type `P` to paste the text above the current line:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
```

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 5
```

Try out some of the other y commands in the table above and get to know the behavior of both the p and P commands. When you are done, return the file to its original state.

Joining Lines

vi is rather strict about its idea of a line. Normally, it is not possible to move the cursor to the end of a line and delete the end-of-line character to join one line with the one below it. Because of this, vi provides a specific command, J (not to be confused with j, which is for cursor movement) to join lines together.

If we place the cursor on line 3 and type the J command, here's what happens:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3 Line 4
Line 5
```


Search-And-Replace

vi has the ability to move the cursor to locations based on searches. It can do this on either a single line or over an entire file. It can also perform text replacements with or without confirmation from the user.

Searching Within A Line

The f command searches a line and moves the cursor to the next instance of a specified character. For example, the command fa would move the cursor to the next occurrence of the character “a” within the current line. After performing a character search within a line, the search may be repeated by typing a semicolon.

Searching The Entire File

To move the cursor to the next occurrence of a word or phrase, the / command is used. This works the same way as we learned earlier in the less program. When you type the / command a “/” will appear at the bottom of the screen. Next, type the word or phrase to be searched for, followed by the Enter key. The cursor will move to the next location containing the search string. A search may be repeated using the previous search string

`q` or `Esc`	Quit substituting.
`l`	Perform this substitution and then quit. Short for “last.”
`Ctrl-e, Ctrl-y`	Scroll down and scroll up, respectively. Useful for viewing the context of the proposed substitution.

If you type `y`, the substitution will be performed, `n` will cause `vi` to skip this instance and move on to the next one.

Editing Multiple Files

It's often useful to edit more than one file at a time. You might need to make changes to multiple files or you may need to copy content from one file into another. With `vi` we can open multiple files for editing by specifying them on the command line:

```
vi file1 file2 file3...
```

Let's exit our existing vi session and create a new file for editing. Type `:wq` to exit `vi`, saving our modified text. Next, we'll create an additional file in our home directory that we can play with. We'll create the file by capturing some output from the `ls` command:


```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Let's edit our old file and our new one with `vi`:

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

`vi` will start up and we will see the first file on the screen:

```
The quick brown fox jumped over the lazy dog. It was cool.
Line 2
Line 3
Line 4
Line 5
```

\033[0;36m	Cyan	\033[1;36m	Light Cyan
\033[0;37m	Light Grey	\033[1;37m	White

Let's try to make a red prompt. We'll insert the escape code at the beginning:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

That works, but notice that all the text that we type after the prompt is also red. To fix this, we will add another escape code to the end of the prompt that tells the terminal emulator to return to the previous color:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

That's better!

It's also possible to set the text background color using the codes listed below. The background colors do not support the bold attribute.

Table 13-3: Escape Sequences Used To Set Background Color

Sequence	Background Color	Sequence	Background Color
\033[0;40m	Black	\033[0;44m	Blue
\033[0;41m	Red	\033[0;45m	Purple
\033[0;42m	Green	\033[0;46m	Cyan
\033[0;43m	Brown	\033[0;47m	Light Grey

We can create a prompt with a red background by applying a simple change to the first escape code:

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

Try out the color codes and see what you can create!

Table 13-5: Breakdown Of Complex Prompt String

Sequence	Action
`\[`	Begins a non-printing character sequence. The purpose of this is to allow `bash` to properly calculate the size of the visible prompt. Without an accurate calculation, command line editing features cannot position the cursor correctly.
`\033[s`	Store the cursor position. This is needed to return to the prompt location after the bar and clock have been drawn at the top of the screen. *Be aware that some terminal emulators do not honor this code.*
`\033[0;0H`	Move the cursor to the upper left corner, which is line 0, column 0.
`\033[0;41m`	Set the background color to red.
`\033[K`	Clear from the current cursor location (the top left corner) to the end of the line. Since the background color is now red, the line is cleared to that color creating our bar. Note that clearing to the end of the line does not change the cursor position, which remains at the upper left corner.
`\033[1;33m`	Set the text color to yellow.
`\t`	Display the current time. While this is a "printing" element, we still include it in the non-printing portion of the prompt, since we don't want `bash` to include the clock when calculating the true size of the displayed prompt.
`\033[0m`	Turn off color. This affects both the text and background.
`\033[u`	Restore the cursor position saved earlier.
`\]`	End the non-printing characters sequence.
`<\u@\h \W>\$`	Prompt string.

Saving The Prompt

Obviously, we don't want to be typing that monster all the time, so we'll want to store our prompt someplace. We can make the prompt permanent by adding it to our `.bashrc` file. To do so, add these two lines to the file:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]
```

Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`yum`

Common Package Management Tasks

There are many operations that 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 discussion below, the term "*package_name*" refers to the actual name of a package rather than the term "*package_file*," which is the name of the file that contains the package.

Finding A Package In A Repository

Using the high-level tools to search repository metadata, a package can be located based on its name or description.

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: To search a `yum` repository for the emacs text editor, this command could be used:

```
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.

Table 14-4: Package Installation Commands

Style	Command(s)
Debian	`apt-get update`

Further Reading

Spend some time getting to know the package management system for your distribution. Each distribution provides documentation for its package management tools. In addition, here are some more generic sources:

- The *Debian GNU/Linux FAQ* chapter on package management provides an overview of package management on Debian systems :
  http://www.debian.org/doc/FAQ/ch-pkgtools.en.html
- The home page for the RPM project:
  http://www.rpm.org
- The home page for the YUM project at Duke University:
  http://linux.duke.edu/projects/yum/
- For a little background, the Wikipedia has an article on metadata:
  http://en.wikipedia.org/wiki/Metadata

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, or 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 try to 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 file system
- `umount` – Unmount a file system
- `fsck` – Check and repair a file system
- `fdisk` – Partition table manipulator
- `mkfs` – Create a file system
- `fdformat` – Format a floppy disk
- `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 tem) has been mounted on `/media/live-1.0.10-8`, and is type iso9660 (a CD-ROM). For 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 file systems 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 file system 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 disk. A mount point is simply a directory somewhere on the file system tree. Nothing special about it. It doesn't even have to be an empty directory, though if you do 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 file system 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? The reason is that 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 multi-tasking), 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 since 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.

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 diskette, and then create a file system. To accomplish the formatting, we use the `fdformat` 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 file system to the diskette with `mkfs`:

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

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

Moving Data Directly To/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. However, 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 are 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
```

Blanking A Re-Writable CD-ROM

Rewritable CD-RW media needs to be erased or *blanked* before it can be 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.

Summing Up

In this chapter we have looked at the basic storage management tasks. There are, of course, many more. Linux supports a vast array of storage devices and file system schemes. It also offers many features for interoperability with other systems.

Further Reading

Take a look at the man pages of the commands we have covered. Some of them support huge numbers of options and operations. Also, look for on-line tutorials for adding hard drives to your Linux system (there are many) and working with optical media.

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 con-

```
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, since 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 it has been determined 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 one 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, to 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:

scp And sftp

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 pathnames 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 only requires 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
```

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 how the logical operators affect its execution:

Test/Action	Is Performed Only If...
`-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.

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
```

thus causing 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 alternate way, using a new feature in `find` itself. We'll talk about the alternate 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, this:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me    me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me    me   0 2008-09-19 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 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me    me   0 2008-09-19 12:53 /home/me/foo.txt
```

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

xargs

The `xargs` command performs an interesting function. It 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 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2008-09-19 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. We'll create a playground and try out some of what we have learned.

First, let's create a playground with lots of subdirectories and files:

nally, 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
```

Alternately, there is a program supplied with `gzip`, called `zcat`, that 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
```

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

bzip2

The `bzip2` program, by Julian Seward, is similar to gzip, but uses a different compression algorithm that 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`:

ern 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

> 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), 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-bar 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` 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 following options:

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$
```

```
mail
news
```

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

paste

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

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

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > dis
tros-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.t
xt
[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`:

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 half way 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:

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

the squeezing will have no effect.

sed

The name `sed` is short for *stream editor*. It performs text editing on a stream of text, ei-

```
Fedora     8     11/08/2007
```

In this example, we print a range of lines, starting with line 1 and continuing to line 5. To do this, we use the `p` command, which simply causes a matched line to be printed. For this to be effective however, we must include the option `-n` (the no auto-print option) to cause `sed` not to print every line by default.

Next, we'll try a regular expression:

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

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

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


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

Here we see the expected result: all 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`. Here 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 only edits lines that match a specified address within the file. The default behavior can be overridden by specifying the `-n` option.
`q`	Exit `sed` without processing any more lines. If the `-n` option is not specified, output the current line.
`Q`	Exit `sed` without processing any more lines.
`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 discussion of *back references* below. 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\}\
```

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 changing the text itself. These tools are often used to prepare text for eventual printing, a subject that we will cover in the next chapter. The programs that we will cover in this chapter include:

- `nl` – Number lines
- `fold` – Wrap each line to a specified length
- `fmt` – A simple text formatter
- `pr` – Prepare 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 files 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, 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:

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 above markup elements must appear alone on its own line. After processing a markup element, `nl` deletes it from the text stream.

Here are 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 `p`*`regexp`* = 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).

	signs negative numbers.
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.

Here are 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 five 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`	By increasing the minimum field width to 10 the padding is now visible.
`380`	`"%+d"`	`+380`	The + flag signs a positive number.
`380`	`"%-d"`	`380`	The - flag left aligns the formatting.

ple 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.*

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.

Tip: 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

`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.

What's important here is that there are no error messages. If there were, the configuration failed, and the program will 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`), that describes the relationships and dependencies among the components that comprise 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 run time.

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
```

turns on the option to highlight search results. Say we search for the word "echo." With this option on, each instance of the word will be highlighted.

`:set tabstop=4`

sets the number of columns occupied by a tab character. The default is 8 columns. Setting the value to 4 (which is a common practice) allows long lines to fit more easily on the screen.

`:set autoindent`

turns on the "auto indent" feature. This causes `vim` to indent a new line the same amount as the line just typed. This speeds up typing on many kinds of programming constructs. To stop indentation, type `Ctrl-d.`

These changes can be made permanent by adding these commands (without the leading colon characters) to your `~/.vimrc` file.

Summing Up

In this first chapter of scripting, we have looked at how scripts are written and made to easily execute on our system. We also saw how we may use various formatting techniques to improve the readability (and thus, the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

Further Reading

- For "Hello World" programs and examples in various programming languages, see:
  http://en.wikipedia.org/wiki/Hello_world
- This Wikipedia article talks more about the shebang mechanism:
  http://en.wikipedia.org/wiki/Shebang_(Unix)

25 – Starting A Project

Starting with this chapter, we will begin to build a program. The purpose of this project is to see how various shell features are used to create programs and, more importantly, create *good* programs.

The program we will write is a *report generator*. It will present various statistics about our system and its status, and will produce this report in HTML format, so we can view it with a web browser such as Firefox or Chrome.

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a very minimal HTML page that contains no system information. That will come later.

First Stage: Minimal Document

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

```
<HTML>
     <HEAD>
           <TITLE>Page Title</TITLE>
     </HEAD>
     <BODY>
           Page body.
     </BODY>
</HTML>
```

If we enter this into our text editor and save the file as `foo.html`, we can use the following URL in Firefox to view the file:

`file:///home/username/foo.html`

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let's start our text editor and create a new file named `~/bin/sys_info_page`:

```
Try `cp --help' for more information.
```

We assign values to two variables, `foo` and `foo1`. We then perform a `cp`, but misspell the name of the second argument. After expansion, the `cp` command is only sent one argument, though it requires two.

There are some rules about variable names:

1. Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
2. The first character of a variable name must be either a letter or an underscore.
3. Spaces and punctuation symbols are not allowed.

The word "variable" implies a value that changes, and in many applications, variables are used this way. However, the variable in our application, `title`, is used as a *constant*. A constant is just like a variable in that it has a name and contains a value. The difference is that the value of a constant does not change. In an application that performs geometric calculations, we might define `PI` as a constant, and assign it the value of `3.1415`, instead of using the number literally throughout our program. The shell makes no distinction between variables and constants; they are mostly for the programmer's convenience. A common convention is to use uppercase letters to designate constants and lower case letters for true variables. We will modify our script to comply with this convention:


```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
        </BODY>
</HTML>"
```

We also took the opportunity to jazz up our title by adding the value of the shell variable `HOSTNAME`. This is the network name of the machine.

Note: The shell actually does provide a way to enforce the immutability of constants, through the use of the `declare` builtin command with the `-r` (read-only) option. Had we assigned `TITLE` this way:

```
declare -r TITLE="Page Title"
```

the shell would prevent any subsequent assignment to `TITLE`. This feature is rarely used, but it exists for very formal scripts.

Assigning Values To Variables And Constants

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

`variable=value`

where *variable* is the name of the variable and *value* is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable; it treats them all as strings. You can force the shell to restrict the assignment to integers by using the `declare` command with the `-i` option, but, like setting variables as read-only, this is rarely done.

Note that in an assignment, there must be no spaces between the variable name, the equals sign, and the value. So what can the value consist of? Anything that we can expand into a string:


```
a=z                     # Assign the string "z" to variable a.
b="a string"            # Embedded spaces must be within quotes.
c="a string and $b"     # Other expansions such as variables can be
                        # expanded into the assignment.
d=$(ls -l foo.txt)      # Results of a command.
e=$((5 * 7))            # Arithmetic expansion.
f="\t\ta string\n"      # Escape sequences such as tabs and newlines.
```

Multiple variable assignments may be done on a single line:

```
a=5 b="a string"
```

During expansion, variable names may be surrounded by optional curly braces “{}”. This is useful in cases where a variable name becomes ambiguous due to its surrounding conmand. There is a third way called a *here document* or *here script*. A here document is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:

```
command << token
text
token
```

where *command* is the name of command that accepts standard input and *token* is a string used to indicate the end of the embedded text. We'll modify our script to use a here document:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIMESTAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIMESTAMP</P>
        </BODY>
</HTML>
_EOF_
```


Instead of using `echo`, our script now uses `cat` and a here document. The string `_EOF_` (meaning "End Of File," a common convention) was selected as the token, and marks the end of the embedded text. Note that the token must appear alone and that there must not be trailing spaces on the line.

So what's the advantage of using a here document? It's mostly the same as `echo`, except that, by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
```

```
     </HEAD>
     <BODY>
           <H1>System Information Report For linuxbox</H1>
           <P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>



     </BODY>
</HTML>
```

we see that there are some blank lines in our output after the timestamp, but we can't be sure of the cause. If we change the functions to include some feedback:

```
report_uptime () {
        echo "Function report_uptime executed."
        return
}

report_disk_space () {
        echo "Function report_disk_space executed."
        return
}

report_home_space () {
        echo "Function report_home_space executed."
        return
}
```


and run the script again:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
     <HEAD>
           <TITLE>System Information Report For linuxbox</TITLE>
     </HEAD>
     <BODY>
           <H1>System Information Report For linuxbox</H1>
           <P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>
           Function report_uptime executed.
           Function report_disk_space executed.
           Function report_home_space executed.
     </BODY>
</HTML>
```

27 – Flow Control: Branching With if

In the last chapter, we were presented with a problem. How can we make our report-generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to "change directions" within our script, based on the results of a test. In programming terms, we need the program to *branch*.

Let's consider a simple example of logic expressed in *pseudocode*, a simulation of a computer language intended for human consumption:

X = 5

If X = 5, then:

    Say "X equals 5."

Otherwise:

    Say "X is not equal to 5."

This is an example of a branch. Based on the condition, "Does X = 5?" do one thing, "Say X equals 5," otherwise do another thing, "Say X is not equal to 5."

if

Using the shell, we can code the logic above as follows:

```
x=5

if [ $x -eq 5 ]; then
     echo "x equals 5."
else
     echo "x does not equal 5."
fi
```

or we can enter it directly at the command line (slightly shortened):

In this example, we execute the `ls` command twice. The first time, the command executes successfully. If we display the value of the parameter `$?`, we see that it is zero. We execute the `ls` command a second time, producing an error, and examine the parameter `$?` again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status,” describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The `true` command always executes successfully and the `false` command always executes unsuccessfully:

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```


We can use these commands to see how the `if` statement works. What the `if` statement really does is evaluate the success or failure of commands:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following `if` executes successfully, and is not executed when the command following `if` does not execute successfully. If a list of commands follows `if`, the last command in the list is evaluated:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

script this way:

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
     if [ $INT -eq 0 ]; then
           echo "INT is zero."
     else
           if [ $INT -lt 0 ]; then
                 echo "INT is negative."
           else
                 echo "INT is positive."
           fi
           if [ $((INT % 2)) -eq 0 ]; then
                 echo "INT is even."
           else
                 echo "INT is odd."
           fi
     fi
else
     echo "INT is not an integer." >&2
     exit 1
fi
```


By applying the regular expression, we are able to limit the value of INT to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

Another added feature of [[ ]] is that the == operator supports pattern matching the same way pathname expansion does. For example:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

This makes [[ ]] useful for evaluating file and pathnames.

nary command, and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We'll discuss `(( ))` and the related arithmetic expansion further in Chapter 34.

Combining Expressions

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 17, when we learned about the `find` command. There are three logical operations for `test` and `[[ ]]`. They are AND, OR and NOT. `test` and `[[ ]]` use different operators to represent these operations :

Table 27-4: Logical Operators

Operation	**test**	**[[ ]] and (( ))**
AND	-a	&&
OR	-o	||
NOT	!	!

Here's an example of an AND operation. The following script determines if an integer is within a range of values:

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
     if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
            echo "$INT is within $MIN_VAL to $MAX_VAL."
     else
            echo "$INT is out of range."
     fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

read – Read Values From Standard Input

The `read` builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

read [-*options*] [*variable*...]

where *options* is one or more of the available options listed below and *variable* is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable `REPLY` contains the line of data.

Basically, `read` assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use `read`, it might look like this:

```
#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then
     if [ $int -eq 0 ]; then
          echo "$int is zero."
     else
          if [ $int -lt 0 ]; then
               echo "$int is negative."
          else
               echo "$int is positive."
          fi
          if [ $((int % 2)) -eq 0 ]; then
               echo "$int is even."
          else
               echo "$int is odd."
          fi
     fi
else
     echo "Input value is not an integer." >&2
     exit 1
fi
```


We use `echo` with the `-n` option (which suppresses the trailing newline on output) to display a prompt, and then use `read` to input a value for the variable `int`. Running this script results in this:

It's possible to supply the user with a default response using the `-e` and `-i` options together:

```
#!/bin/bash

# read-default: supply a default value if user presses Enter key.

read -e -p "What is your user name? " -i $USER
echo "You answered: '$REPLY'"
```

In this script, we prompt the user to enter his/her user name and use the environment variable `USER` to provide a default value. When the script is run it displays the default string and if the user simply presses the Enter key, `read` will assign the default string to the `REPLY` variable.

```
[me@linuxbox ~]$ read-default
What is your user name? me
You answered: 'me'
```


IFS

Normally, the shell performs word splitting on the input provided to `read`. As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line, and are assigned to separate variables by `read`. This behavior is configured by a shell variable named `IFS` (for Internal Field Separator). The default value of `IFS` contains a space, a tab, and a newline character, each of which will separate items from one another.

We can adjust the value of `IFS` to control the separation of fields input to `read`. For example, the `/etc/passwd` file contains lines of data that use the colon character as a field separator. By changing the value of `IFS` to a single colon, we can use `read` to input the contents of `/etc/passwd` and successfully separate fields into different variables. Here we have a script that does just that:

```
#!/bin/bash

# read-ifs: read fields from a file

FILE=/etc/passwd
```

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=

if [ $number = 1 ]; then
     echo "Number is equal to 1."
else
     echo "Number is not equal to 1."
fi
```

Running the script with this change results in the output:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
 Number is not equal to 1.
```

We get this rather cryptic error message, followed by the output of the second `echo` command. The problem is the expansion of the `number` variable within the `test` command. When the command:

```
[ $number = 1 ]
```

undergoes expansion with `number` being empty, the result is this:

```
[  = 1 ]
```

which is invalid and the error is generated. The `=` operator is a binary operator (it requires a value on each side), but the first value is missing, so the `test` command expects a unary operator (such as `-z`) instead. Further, since the `test` failed (because of the error), the `if` command receives a non-zero exit code and acts accordingly, and the second `echo` command is executed.

This problem can be corrected by adding quotes around the first argument in the `test` command:

```
[ "$number" = 1 ]
```

terminate each action, so now we can do this:

```
#!/bin/bash

# case4-2: test a character

read -n 1 -p "Type a character > "
echo
case $REPLY in
    [[:upper:]])    echo "'$REPLY' is upper case." ;;&
    [[:lower:]])    echo "'$REPLY' is lower case." ;;&
    [[:alpha:]])    echo "'$REPLY' is alphabetic." ;;&
    [[:digit:]])    echo "'$REPLY' is a digit." ;;&
    [[:graph:]])    echo "'$REPLY' is a visible character." ;;&
    [[:punct:]])    echo "'$REPLY' is a punctuation symbol." ;;&
    [[:space:]])    echo "'$REPLY' is a whitespace character." ;;&
    [[:xdigit:]])   echo "'$REPLY' is a hexadecimal digit." ;;&
esac
```

When we run this script, we get this:

```
[me@linuxbox ~]$ case4-2
Type a character > a
'a' is lower case.
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```


The addition of the ";;&" syntax allows case to continue on to the next test rather than simply terminating.

Summing Up

The `case` command is a handy addition to our bag of programming tricks. As we will see in the next chapter, it's the perfect tool for handling certain types of problems.

Further Reading

- The *Bash Reference Manual* section on Conditional Constructs describes the `case` command in detail:
  http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21
- The *Advanced Bash-Scripting Guide* provides further examples of `case` applica-

tions:
http://tldp.org/LDP/abs/html/testbranch.html

```
interactive=
filename=

while [[ -n $1 ]]; do
     case $1 in
           -f | --file)            shift
                                   filename=$1
                                   ;;
           -i | --interactive)     interactive=1
                                   ;;
           -h | --help)            usage
                                   exit
                                   ;;
           *)                      usage >&2
                                   exit 1
                                   ;;
     esac
     shift
done

# interactive mode

if [[ -n $interactive ]]; then
     while true; do
           read -p "Enter name of output file: " filename
           if [[ -e $filename ]]; then
                 read -p "'$filename' exists. Overwrite? [y/n/q] > "
                 case $REPLY in
                       Y|y)  break
                             ;;
                       Q|q)  echo "Program terminated."
                             exit
                             ;;
                       *)    continue
                             ;;
                 esac
           fi
     done
fi

# output html page

if [[ -n $filename ]]; then
     if touch $filename && [[ -f $filename ]]; then
           write_html_page > $filename
     else
           echo "$PROGNAME: Cannot write file '$filename'" >&2
           exit 1
     fi
else
```

The really powerful feature of `for` is the number of interesting ways we can create the list of words. For example, through brace expansion:

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

or pathname expansion:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```


or command substitution:

```
#!/bin/bash

# longest-word : find longest string in a file

while [[ -n $1 ]]; do
     if [[ -r $1 ]]; then
            max_word=
            max_len=0
            for i in $(strings $1); do
                  len=$(echo $i | wc -c)
                  if (( len > max_len )); then
                        max_len=$len
                        max_word=$i
                  fi
            done
            echo "$1: '$max_word' ($max_len characters)"
     fi
     shift
done
```

```
                        max_len=$len
                        max_word=$j
                  fi
            done
            echo "$i: '$max_word' ($max_len characters)"
      fi
      shift
done
```

Next, we will compare the efficiency of the two versions by using the `time` command:

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real 0m3.618s
user 0m1.544s
sys  0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real 0m0.060s
user 0m0.056s
sys  0m0.008s
```


The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds—a very significant improvement.

Case Conversion

Recent versions of `bash` have support for upper/lowercase conversion of strings. `bash` has four parameter expansions and two options to the `declare` command to support it.

So what is case conversion good for? Aside from the obvious aesthetic value, it has an important role in programming. Let's consider the case of a database look-up. Imagine that a user has entered a string into a data input field that we want to look up in a database. It's possible the user will enter the value in all uppercase letters or lowercase letters or a combination of both. We certainly don't want to populate our database with every possible permutation of upper and lower case spellings. What to do?

A common approach to this problem is to *normalize* the user's input. That is, convert it into a standardized form before we attempt the database look-up. We can do this by con-

<<	Left bitwise shift. Shift all the bits in a number to the left.
>>	Right bitwise shift. Shift all the bits in a number to the right.
&	Bitwise AND. Perform an AND operation on all the bits in two numbers.
|	Bitwise OR. Perform an OR operation on all the bits in two numbers.
^	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers.

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.

Here we will demonstrate producing a list of powers of 2, using the left bitwise shift operator:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```


Logic

As we discovered in Chapter 27, the `(( ))` compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Here is the complete list:

Table 34-6: Comparison Operators

Operator	Description
<=	Less than or equal to
>=	Greater than or equal to
<	Less than
>	Greater than

`==`	Equal to
`!=`	Not equal to
`&&`	Logical AND
`||`	Logical OR
`expr1?expr2:expr3`	Comparison (ternary) operator. If expression *expr1* evaluates to be non-zero (arithmetic true) then *expr2*, else *expr3*.

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The `(( ))` compound command maps the results into the shell's normal exit codes:

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```


The strangest of the logical operators is the *ternary operator*. This operator (which is modeled after the one in the C programming language) performs a standalone logical test. It can be used as a kind of if/then/else statement. It acts on three arithmetic expressions (strings won't work), and if the first expression is true (or non-zero) the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable `a` switches from zero to one or vice versa.

Please note that performing assignment within the expressions is not straightforward.

```
[5]=Fri [6]=Sat)
```

Accessing Array Elements

So what are arrays good for? Just as many data-management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays.

Let's consider a simple data-gathering and presentation example. We will construct a script that examines the modification times of the files in a specified directory. From this data, our script will output a table showing at what hour of the day the files were last modified. Such a script could be used to determine when a system is most active. This script, called `hours`, produces this result:

```
[me@linuxbox ~]$ hours .
Hour Files Hour  Files
---- ----- ----  -----
00   0     12    11
01   1     13    7
02   0     14    1
03   0     15    7
04   1     16    6
05   1     17    5
06   6     18    4
07   3     19    4
08   1     20    1
09   14    21    0
10   2     22    0
11   5     23    0

Total files = 80
```


We execute the `hours` program, specifying the current directory as the target. It produces a table showing, for each hour of the day (0-23), how many files were last modified. The code to produce this is as follows:

```
#!/bin/bash

# hours : script to count files by modification time

usage () {
     echo "usage: $(basename $0) directory" >&2
}
```

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

Sorting An Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:

```
#!/bin/bash

# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array:   ${a_sorted[@]}"
```


When executed, the script produces this:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:   a b c d e f
```

The script operates by copying the contents of the original array (`a`) into a second array (`a_sorted`) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

Deleting An Array

To delete an array, use the `unset` command:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

array[@]}" expansion which expands into the list of array indexes rather than the list of array elements.

Process Substitution

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means that the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) is lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

We saw an example of the subshell environment problem in Chapter 28, when we discovered that a `read` command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

```
echo "foo" | read
echo $REPLY
```


The contents of the REPLY variable is always empty because the `read` command is executed in a subshell, and its copy of REPLY is destroyed when the subshell terminates.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called *process substitution* that can be used to work around this problem.

Process substitution is expressed in two ways:

For processes that produce standard output:

`<(`*`list`*`)`

or, for processes that intake standard input:

`>(`*`list`*`)`

where *list* is a list of commands.

To solve our problem with `read`, we can employ process substitution like this:

```
read < <(echo "foo")
echo $REPLY
```

```
for i in {1..5}; do
     echo "Iteration $i of 5"
     sleep 5
done
```

This script features two `trap` commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an `exit` command in each of the signal-handling functions. Without an `exit`, the script would continue after completing the function.

When the user presses `Ctrl-c` during the execution of this script, the results look like this:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```


Temporary Files

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the `/tmp` directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files non-predictable filenames. This avoids an exploit known as a *temp race attack*. One way to create a non-predictable (but still descriptive) name is to do something like this:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

This will create a filename consisting of the program's name, followed by its process ID (PID), followed by a random integer. Note, however, that the `$RANDOM` shell variable only returns a value in the range of 1-32767, which is not a very large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

A better way is to use the `mktemp` program (not to be confused with the `mktemp` standard library function) to both name and create the temporary file. The `mktemp` program accepts a template as an argument that is used to build the filename. The template should include a series of "X" characters, which are replaced by a corresponding number of random letters and numbers. The longer the series of "X" characters, the longer the series of random characters. Here is an example:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

This creates a temporary file and assigns its name to the variable `tempfile`. The "X" characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter `$$` to obtain the PID) might be something like:

```
/tmp/foobar.6593.UOZuvM6654
```

For scripts that are executed by regular users, it may be wise to avoid the use of the `/tmp` directory and create a directory for temporary files within the user's home directory, with a line of code such as this:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```


Asynchronous Execution

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multiuser as well. Scripts can be constructed to behave in a multitasking fashion.

Usually this involves launching a script that, in turn, launches one or more child scripts that perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other, and one script must wait for the other to finish its task before finishing its own?

`bash` has a builtin command to help manage *asynchronous execution* such as this. The `wait` command causes a parent script to pause until a specified process (i.e., the child script) finishes.

wait

We will demonstrate the `wait` command first. To do this, we will need two scripts, a parent script:

- *The Advanced Bash-Scripting Guide* also has a discussion of process substitution: http://tldp.org/LDP/abs/html/process-sub.html
- *Linux Journal* has two good articles on named pipes. The first, from September 1997:
  http://www.linuxjournal.com/article/2156
- and the second, from March 2009:
  http://www.linuxjournal.com/content/using-named-pipes-fifos-bash

Index