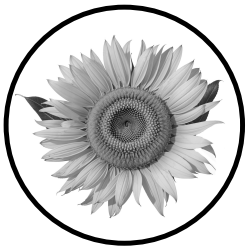
WORKING IN THE COMMAND INTERFACE



When you work in a graphical user interface (GUI), you usually start up your program or application by clicking or double-clicking on an icon, or selecting its name from a menu. In a command line interface, you can't do that. Instead, you have to enter the name of the program you want to run along with any arguments it may require, followed by the ENTER key. The command line interface then takes the steps necessary to run your program. The command line interface in Unix is called a *shell* and one of the most popular shells on Linux systems is `bash`.

The purpose of this chapter is to give you enough of an introduction to working in a shell, `bash` in particular, so that you're able to understand the rest of the book. It's intended to help those readers who have little experience working in `bash`; however, even if you're already comfortable working in `bash` or some other shell, you still might benefit from reading this because it may shed some light on things you thought you understood well.

The chapter is intentionally incomplete, covering only what I think you need to know to get started. I begin with a brief history of Unix shells and then describe the essential features of bash and how to take advantage of many of them, including I/O redirection, background processing, shell parameters and variables, file globs, control flow, command substitution, environment variables, scripting, and the history mechanism. I don't cover all of its features. For example, I've omitted discussion of the various modes in which you can run bash as well as details about its use as a programming language. Instead I concentrate on the most essential and useful features of the shell.

After this exploration of the shell, I present the most common and useful commands that you'll need. This includes a discussion of the types of commands and how to view and change file permissions. We'll primarily explore the basic commands for file and directory manipulation. Lastly, I include a brief introduction to filters and regular expressions.

My hope is that, after you've read what I've described here, you'll be interested enough to learn more on your own.

Commands Are (Usually) Not Part of the Shell

Let's clarify the difference between the shell and the commands that you enter when you work in it. The shell is essentially an interpreter program that displays a prompt and then waits for you to enter a line of text, meaning text followed by a newline character, which you input by pressing ENTER. You're supposed to enter a command at the prompt, not some arbitrary text. A command is typically the name of a program. For example, in bash, an interaction might look like this:

```
$ hostname
harpo
$
```

The $ character is the shell's prompt and hostname is the command that I entered. I pressed ENTER immediately after hostname. The output of this command is the string harpo, which is the name I gave to the computer on which I was working. After the output appears on my terminal, bash redisplays the prompt so that I can enter another command. The hostname program is actually a file in the filesystem; it isn't part of bash. I can run this program from any other shell and it will produce the same output.

Most commands are just programs that live in the filesystem—they can be run in any shell. Learning to work in a command line environment consists of two separate tasks:

- Learning how to use the shell to improve your efficiency and save you lots of time
- Learning what the various commands are, what they do, and how to use them

The rest of this chapter is designed to help you accomplish both of these tasks. I'll start with a tutorial on bash.

Historical Remarks About Shells

You'll have a better understanding of bash and other shells if you understand their origins and how features were incorporated into them over the years.

Ken Thompson wrote the shell for the first version of Unix in 1969. He borrowed ideas from the shell used in the MULTICS project, on which he had worked. The term *shell* had been used in that project as another name for a command line interpreter. The original Thompson shell lacked many features of modern day shells.

The C shell was written by Bill Joy at UC Berkeley and made its appearance in Sixth Edition Unix (1975), the first widely distributed release from UC Berkeley. It was an enhancement of the original Thompson shell with C-like syntax. Among its notable additions are the history mechanism and command line editing, and it's part of all BSD distributions.

The Bourne shell, written by Stephen Bourne, was introduced into Unix in System 7 (1979), which was the last release of Unix by AT&T Bell Labs prior to the commercialization of UNIX by AT&T. Its syntax derives in part from the Algol 68 programming language and is a part of all Unix releases.

David Korn developed the Korn shell at Bell Labs and introduced it into the SVR4 commercial release of Unix by AT&T. Its syntax is based on the Bourne shell, but it had many more features.

The TENEX C shell, or TC shell, extended the C shell with command line editing and completion, as well as other features which were found in the TENEX operating system. It was written by Ken Greer and others in 1983.

The Bourne Again SHell (bash) is an extension of the Bourne shell with many of the sophisticated features of the Korn shell and the TC shell. It's the default user shell in Linux and appears at the top of most lists of the best or most popular Unix shells.

Many Linux distributions have incorporated a new shell known as dash. The dash shell is a POSIX-conformant implementation of the Bourne shell that is designed to be fast and efficient. It's a direct descendant of the Almquist SHell (ash), ported to Linux in 1997, and it was renamed dash in 2002.

Working with the Shell

Your view and appreciation of Unix is pretty much determined by the interface that the shell creates for you. The shell hides the inner workings of the kernel, presenting a set of high-level functions that can make the system easy to use. It's like the walnut pictured in Figure 1; the shell hides the kernel in the same way.

Figure 1: The shell hides the kernel of Unix.

If you've used a Unix system with a graphical user interface, be aware that this GUI is a separate and distinct application with respect to the Unix operating system. The GUI provides an alternative to a shell for interacting with Unix, but experienced users typically do most of their work in a shell, perhaps because it's much faster to type on a keyboard than it is to point and click with a mouse or its equivalent. In a terminal window running a shell, we have easy access to hundreds of commands.

Unlike most operating systems, Unix allows you to replace the default login shell with one of your own choosing. The particular shell that will run when you log in is specified in an entry for your user account in a file called the *password file*. On all systems, the system administrator can change this entry, and on some, you can change it yourself with the `chsh` command. When you enter `chsh` at the shell prompt, you'll either see the message `Command not found` or be prompted for your password. As an alternative on some versions of Unix, you can enter `passwd -s` to change your login shell. In short, you might need someone with administrative privileges to change your login shell on some systems.

Shell Features

In all shells, a simple command is of the following form:

```
command-name command-options arg1 arg2 ... argn
```

The shell waits for you to enter a newline character (by pressing ENTER) before it attempts to interpret (or *parse*) a command. A newline signals the end of the command.

Once the shell receives the entered line, it checks to see whether *`command-name`* is a builtin command. A *builtin command* is a command whose implementation is hardcoded into the shell itself. For example, `help` is a builtin command in `bash`; when you enter `help`, `bash` displays helpful information. If the command is a shell builtin, the shell itself executes the command. (I explain more about builtin commands in "Types of Commands" on page 30.) If it isn't, the shell searches for a file whose pathname, either relative or absolute, is *`command-name`*. I'll explain how that search takes place in "Parameters and Variables" on page 15.

If the shell doesn't find the file, it displays a message such as *`command-name`*: `command not found` and redisplays the prompt. If it finds the file, it creates a

new process to execute the command, passing it the arguments from the command line. The new process executes the command and, when the command terminates, the created process terminates also. In the meantime, the shell does nothing other than wait for this process to finish executing. Only then does the shell display its prompt so that you can enter another command.

In addition to command interpretation, all shells provide the following:

Redirection of the input and output of commands Changing the source of input or the target of output, respectively

Pipes A method of channeling the output of one command to the input of another

Scripting Storing a sequence of shell instructions and commands in a file that can be executed

Filename substitution using metacharacters A method of naming one or more files using a pattern-matching language that has wildcards and other operators

Control flow constructs Loops and conditional execution

Shells written after the Bourne shell also provide the following:

History mechanism A method of saving and reissuing commands in whole or in part using a succinct notation

Backgrounding and job control A method of controlling the sequencing and timing of commands

User-defined aliases Typically for frequently used commands

The bash shell has a number of features in addition to these, including:

- Interactive file and command name completion
- General interactive command line editing
- Coprocesses: named processes that run concurrently with the shell
- More general redirection of input and output of commands
- Array variables within the shell
- Autoloading of function definitions from files
- Restricted shells: shells that limit what a user can do, for security reasons

In the following sections, we'll explore some of these features in more depth.

Standard I/O and Redirection

Unix uses a clever method of handling I/O. Every program is automatically given three open input/output streams when it begins execution: *standard input*, *standard output*, and *standard error*. They're called *streams* because they're linear sequences of characters. By default, standard input comes from the keyboard, and standard output and error are written to the

terminal window. The shell can reference these streams using the numbers 0, 1, and 2 for standard input, standard output, and standard error, respectively, as illustrated in Figure 2.

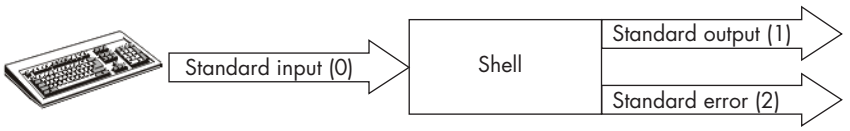


Figure 2: The three streams assigned to every process

Commands usually read from standard input, write to standard output, and send their error messages to the standard error stream, which appears in the terminal window. The shell, however, can trick a command into reading from a different source or writing to a different source. This is called *I/O redirection*. For example, the command

```
$ ls mydir
```

lists the files in the directory named *mydir* on the terminal. In contrast, the command

```
$ ls mydir > mylisting
```

creates a file called *mylisting* in the current working directory and redirects the output of the `ls` command to *mylisting*, provided that *mylisting* doesn't already exist. If it does exist, `bash` usually displays a message such as the following:

```
bash: mylisting: cannot overwrite existing file
```

The greater-than character (`>`) in the preceding example is the shell *output redirection operator*. It replaces the standard output of the program to which it is applied with the file whose name immediately follows the operator. This means `> outfile` writes the standard output of the command to which you apply it to a file named *outfile* instead of to the terminal, provided that either the file doesn't exist or that `bash` is allowed to overwrite the file.

Controlling Overwriting of Files

System administrators usually configure users' shells so that their default behavior is to prevent files from being overwritten. They do this by including either of the following two lines in the shell's startup file to enable a shell option named `noclobber`:

```
set -o noclobber
set -C          # Short form of set -o noclobber
```

In bash, the startup file is *.bashrc*, located in the home directory. Other shells have startup files with different names. If noclobber is set, overwriting the file is prevented; otherwise, the redirected output replaces the file's contents.

Enter echo $SHELLOPTS in your shell to check whether noclobber is set. If noclobber appears in the output, it's set; otherwise, it isn't. The following output shows that noclobber is set:

```
$ echo $SHELLOPTS
braceexpand:hashall:history:ignoreeof:interactive-comments:monitor:noclobber
```

If it isn't set in your shell, in bash you can set it yourself by editing your *.bashrc* to include either of the two lines shown in the previous listing.

If you're willing to overwrite a file even though noclobber is set, use the >| operator, as in the following:

```
$ ls mydir >| mylisting
```

This command redirects standard output to a file, overwriting it if it already exists, ignoring noclobber. Lastly, you can unset noclobber by entering set +o noclobber, which unsets it in the terminal in which this shell is running.

Redirecting Standard Input

The notation < infile means "read the input from the file *infile* instead of from the standard input stream, which is by default the keyboard." The < operator is called the *input redirection operator*. If a program named cmd expects input from the keyboard, the command cmd < infile replaces the keyboard with the contents of *infile*.

To demonstrate, suppose we've written a program named sum that expects the user to enter numbers on the keyboard, one per line, ending by pressing CTRL-D, and then prints their sum:

```
$ sum
1
2
3
4
CTRL-D entered here
10
```

If *numbers* is a file containing these same four numbers, one per line, we can enter the following:

```
$ sum < numbers
10
```

The input to sum came from *numbers* instead of the keyboard.

Redirecting Both Standard Input and Output

We can redirect both the input and output of a command:

```
$ command < infile > outfile
```

This causes command to read its input from *infile* and send its output to *outfile*.

The order of the the input and output redirection operators doesn't matter. For example, we can enter

```
$ sum < numbers > total
$
```

or

```
$ sum > total < numbers
$
```

and the sum of the numbers will be written to the file named *total* instead of to the screen.

In bash, we can write any of the following semantically equivalent lines:

```
$ command > outfile < infile
$ > outfile command < infile
$ < infile > outfile command
```

Only the first of these is POSIX-compliant syntax, and it's best to avoid putting any redirection before the command name.

Appending Redirected Output

The *append operator* (>>) is like the output redirection operator, but instead of overwriting the target file, it *appends* the output to the end of it. One example of its use is in maintaining logfiles. If you have a file named *mylog* in your current working directory that already has log data in it and you run the command

```
$ echo 'Today I learned about redirection' >> mylog
```

then that line will be appended to *mylog*, and if *mylog* didn't exist, it will be created with that text as its first line.

Redirecting with Pipes

We can carry the concept of redirection one step further to allow the output of one command to be the input of another. This is known as a *pipe* or *pipeline*, and the operator that does it is a vertical bar (|). For example:

```
$ ls mydir | sort | lpr
```

This pipeline makes the output of `ls mydir` become the input to the `sort` command, which then sorts it, and sends the sorted list to the `lpr` command, which is a command to send files to the default printer attached to the system.

We could accomplish the same result using temporary files rather than pipes, provided that *temp1* and *temp2* don't already exist, as follows:

```
$ ls mydir > temp1
$ sort < temp1 > temp2
$ lpr < temp2
$ rm temp1 temp2
```

It's not exactly the same, because when a pipeline is established, the commands run simultaneously. In the example using the pipe, the `ls`, `sort`, and `lpr` commands start up together. As `ls` is busy working and producing some output, `sort` is receiving that output. The shell uses a kernel mechanism called a *pipe* to implement this communication. (We explore kernel pipes in Chapter 13 of the book.) The `bash` shell provides a few more redirection operators such as `<<` and `<<<`, but I don't explain them here. The `bash` man page describes them well.

Using Redirection for Error Handling

Neither the output redirection operator nor the append operator sends the standard error stream to the given file. This means that you'll see any error messages sent to the standard error stream on your terminal even if you redirect output.

Some commands can produce many error messages, and sometimes you don't care about the errors. For example, if you try to list every file in the entire filesystem using `ls -R /`, you'll see many `Permission denied` errors.

The `find` command can also generate many error messages. You can use `find` to search through a part of the directory hierarchy for files that satisfy various criteria, and you can run commands on those files when it finds them. Sometimes you'll get lots of errors because you don't have permission to see the content of certain directories. For example:

```
$ find /var -name "*.log"
find: '/var/spool/rsyslog': Permission denied
find: '/var/spool/cups': Permission denied
find: '/var/spool/cron/atjobs': Permission denied
find: '/var/spool/cron/crontabs': Permission denied
find: '/var/spool/cron/atspool': Permission denied
--snip--
```

Here, `find` begins a search of the directory hierarchy rooted in the top-level directory */var* for files whose names end in *.log*, and it prints their pathnames if it finds any. To specify this set of files, we use the `bash` *wildcard* operator (*), which matches any sequence of zero or more characters that does not start with a period. Because * matches any string of characters, the expression `*.log` matches filenames that begin with any non-period character and end in *.log*. (See "File Globs" on page 17 for a description of this type of pattern matching.)

We can prevent all error messages from appearing on the screen by redirecting them with the bash construct `2>/dev/null`. The `2>` part of this construct tells bash to send the standard error stream, whose number is 2, into the file */dev/null*, but */dev/null* is not a real file; it's like a black hole in that anything you write to it disappears. (The kernel discards it.) What follows is part of what we see when we redirect the error stream to */dev/null* with the previous command:

```
$ find /var -name "*.log" 2>/dev/null
/var/lib/texmf/web2c/metafont/mf.log
/var/lib/texmf/web2c/tex/tex.log
/var/lib/texmf/web2c/updmap.log
/var/lib/texmf/web2c/luatex/lualatex.log
/var/lib/texmf/web2c/luatex/lualatex-dev.log
--snip--
```

If we want to redirect both the standard output and the standard error streams to the same file, we'd use the `&>` operator, as in

```
$ find /var -name "*.log" &> myfile
```

which sends both streams to the file *myfile*. Similarly, to append standard output and standard error of the preceding command to *myfile*, we'd write:

```
$ find /var -name "*.log" &>> myfile
```

To learn more about I/O redirection, including several operators not described here, read the bash man page.

Control Operators and Multitasking

You can enter multiple commands on a single line by using certain control operators that are usually called *command separators*. Command separators are characters used to terminate commands that appear on the same line.

Sequencing

The semicolon (`;`) acts like a newline character to the shell; it terminates the preceding command:

```
$ echo 'hello world' ; hostname ; whoami
hello world
harpo
stewart
```

This example runs each of the three commands, one after the other. It also introduces the whoami command, which displays the username of the user running it. The `;` is used to sequentially execute the commands.

Grouping Commands

You can use parentheses to group a sequence of commands so that they are treated like a single command. We often need to do this to bypass the operator precedence rules of the shell. To demonstrate:

```
$ echo 'hello world' ; hostname; whoami > outfile
hello world
harpo
$
```

The output of `echo 'hello world' ; hostname` appears on the screen and only the output of `whoami` is written to the file named *outfile*. This is because the semicolon has higher precedence than the redirection operator, which applies only to the `whoami` command. Now try this:

```
$ echo 'hello world' ; (hostname; whoami) > outfile
hello world
$
```

In this case, the output of `hostname ; whoami` is written to the file. Grouping parentheses comes in handy in many situations.

Backgrounding and Job Control

The ampersand (`&`) is another useful control operator. When a command is terminated with `&`, the shell doesn't wait for the command to finish before it redisplays the prompt, which is useful when we run commands that take a long time but we want to continue working. In this case, we say that the command, or *job*, is running *in the background*.

We can't see the effect of using the `&` unless we've got a long-running command, but we can simulate a long-running command with the `sleep` command, which simply puts the calling shell to sleep for the number of seconds that we provide as an argument. In other words, `sleep` doesn't terminate until the amount of time elapses:

```
$ sleep 10
10 seconds pass here
$
```

This command makes the shell wait for 10 seconds before the prompt returns.

The following command prints `done sleeping` after 10 seconds:

```
$ sleep 10 ; echo done sleeping
10 seconds pass here
done sleeping
$
```

If we terminate this entire command with the &, the prompt returns immediately after printing the job number and the process ID (PID) of the job it started in the background:

```
$ (sleep 10 ; echo done sleeping) &
[1] 15340
$ done sleeping
```

Here it prints [1] 15340 to indicate that the backgrounded job is job number 1 and has PID 15340.

To use the job number to refer to the job, precede the number with a percent sign (%). For example, %1 refers to job number 1.

When a command is backgrounded, by default its output still goes to the terminal, which is why the done sleeping message appears after the prompt. We can avoid this behavior by redirecting the output into a file:

```
$ echo 'hello world' > outfile &
$
```

This example causes the echo command to work in the background, writing hello world into the file *outfile*.

Backgrounding is a form of *multitasking*. It allows you to run more than one process under the same shell at the same time. The & doesn't need to be at the end of the line; we can also use it to run separate commands simultaneously by putting it between them:

```
$ echo hello world > outfile1 & whoami > outfile2 &
```

This tells the shell to run the echo and whoami commands *concurrently* and in the background, putting their outputs into the *outfile1* and *outfile2* files, respectively.

The > operator has higher precedence than the & operator, which means that the previous commands are equivalent to the following:

```
$ (echo hello world > outfile1) & (whoami > outfile2) &
```

Suppose that you put a job in the background and some time later you want to run it in the foreground. The fg command (for *foreground*) will bring a job to the foreground. Its general form is fg *job-spec*, where a *job-spec* is (%) followed by a number, a string, or a question mark (?) followed by a string:

fg %*n* Foregrounds job *n*

fg %*string* Foregrounds the job whose command begins with *string*

fg %?*string* Foregrounds the job whose command contains *string*

If there's only a single job running in the background, you can omit the *job-spec* entirely and just enter fg to bring that job to the foreground. In all cases, the command that you entered and put in the background is written to the terminal when it is brought to the foreground.

Let's go over some examples. Assume the backgrounded command is the following:

```
(sleep 30 ; echo done sleeping) &
```

If this is the only job running in the background, just enter `fg`:

```
$ (sleep 30 ; echo done sleeping) &
[1] 15412
$ fg
(sleep 30; echo done sleeping)
time elapses
$
```

You can also enter the `fg` command explicitly:

```
$ (sleep 30 ; echo done sleeping) &
[1] 15373
$ fg %1
(sleep 30; echo done sleeping)
time elapses
$
```

You can use any prefix of the command name after the `%` to refer to the job, but if it contains special characters such as `(`, you need to enclose it in single quotes:

```
$ (sleep 30 ; echo done sleeping) &
[1] 15430
$ fg %'( sle'
(sleep 30; echo done sleeping)
time elapses
$
```

If multiple jobs start with that prefix, you'll get an error message.

In the third method, you can choose any substring contained in the command instead of a prefix of it with the `%?` operator:

```
$ (sleep 30 ; echo done sleeping) &
[1] 15430
$ fg %?echo
( sleep 30; echo done sleeping )
time elapses
$
```

The same rules apply about enclosing it in single quotes.

Finally, you can omit the `fg` entirely and just enter a *job-spec* to bring the job to the foreground. In the preceding example, we could have just entered:

```
$ (sleep 30 ; echo done sleeping) &
[1] 15430
```

```
$ %?echo
( sleep 30; echo done sleeping )
time elapses
$
```

If you ran a command in the foreground and decide you want to put it into the background so you can work on something else while it's running, you can follow this two-step process:

1. Enter the *suspend character* by pressing CTRL-Z to stop but not terminate the running command. This is called *suspending* the process.
2. Enter the *background* command, `bg`, to put it into the background.

These operators are usually sufficient for most tasks, but if you're curious, see the `JOB CONTROL` section of the `bash` man page to learn more.

The Shell as a Command

You can run a shell the same way that you run commands. Enter the name of a shell—for example, `bash`, `sh`, `csh`, and so on— at the command prompt within any shell to start another shell:

```
$ bash
$
```

In this example, we start the `bash` shell. If you do this, you'll have two instances of the `bash` shell running, but the first will be dormant, waiting for the second to exit, and the second will become your active shell. When one shell is created as a result of a command given in another shell, the created shell is called the *subshell* of the original shell, which is called the *parent shell*.

If you forget which shell your currently active shell is (which can happen when the prompt doesn't change), two simple commands can tell you. The first one is as follows:

```
$ ps -p $$
    PID TTY          TIME CMD
   9855 pts/1    00:00:00 bash
```

The `ps` command prints information about some, but not all, processes that you own. The `-p` option is followed by the process IDs of processes whose information you want to see. Most shells store their process IDs in a variable named `$`. By preceding the PID with a `$`, you're telling the shell to evaluate the variable and replace it with that value before passing the argument to the command. In this example, `ps -p` has the argument `$$`, so the shell evaluates it and passes the process ID 9855 to `ps`.

The following shows a second method to see which shell is active:

```
$ echo $0
bash
```

In the Introduction to the book, I used the `echo` command to demonstrate some notation. That command might have seemed rather useless, but it's very convenient. The `echo` command displays its arguments. If an argument is a variable whose name is preceded by a `$`, the current shell evaluates the variable and replaces it with its value. Therefore, if you give `echo` one or more variable names preceded by `$`s, you'll see the values of those variables. The shell variable `0` stores the name of the currently active shell, so `echo $0` displays its name.

As another example of the use of `echo`, you can display the value of any environment variable with it; for example, to display the value of the `SHELL` environment variable, you can enter:

```
$ echo $SHELL
/bin/bash
```

Note that `SHELL` is the name of your *login shell*, which is the one started up when you logged in, so in this example, you see your login shell's name, which may not be the same as that of your currently active shell.

Shell Parameters and Variables

Because `bash` is a full-fledged programming language, it has many features similar to higher-level compiled languages, including parameters. As the `bash` man page states, a *parameter* is an entity that stores a value. It's a generalization of the idea of a variable. Parameters are denoted by one of the following:

Name A string beginning with a letter or underscore and containing only letters, digits, and underscores, such as `user_name`

Number Any sequence of decimal digits, such as 12, excluding 0

Special character One of the following characters: `@`, `*`, `?`, `-`, `#`, `$`, `!`, or `0`

When a parameter is identified by a name, it's called a *variable*. Whereas users can assign a value to a variable, only `bash` can give a value to a parameter that isn't a variable. We'll discuss the use of some `bash` parameters in "Shell Scripts" on page 23.

Some variables are built into `bash`; these are called *shell variables*. In general, you shouldn't alter the values of most shell variables, since `bash` uses them in specific ways. For example, `PWD` is the pathname of the current working directory and you shouldn't change it. Some shell variables, though, are customizable, such as `PATH`, which is the search path for finding commands. It's a colon-separated list of absolute pathnames of directories in which `bash` looks for commands, such as:

```
/bin:/usr/local/bin:/usr/bin
```

When you enter a command whose name doesn't contain a slash character, such as `echo`, `bash` searches this list of directory paths in left-to-right order to try to find the matching command. Specifically, for each directory pathname

in the list, it appends the command name to the end of that pathname, preceded by a slash, and checks whether it's an executable file that you have permission to execute. It does this until it finds one or has searched the entire list of pathnames in PATH.

For example, with the PATH shown previously, if you enter the echo command, it checks whether */bin/echo* exists and is executable by you, then whether */usr/local/bin/echo* exists and is executable by you, and finally it checks */usr/bin/echo*, which is the echo command that is executed for you. You can modify the PATH variable to customize your search path. You can rearrange the order of its directories and add and remove directories from it.

You can also create your own variables with an assignment operator:

```
$ mygreeting=hello
```

This creates and initializes a variable named mygreeting. Very often, program developers create environment variables that allow the user to control the behavior of the program. In Chapter 3 of the book, you'll see examples of commands that use environment variables to customize their output.

When a variable is given a value, we say that it's *set*, even if it's a null string. The only way to remove a variable is to *unset* it with the unset builtin bash command:

```
$ unset mygreeting
```

Parameters and variables are evaluated when their names are preceded by a $. This is called *parameter expansion*. Here's another example showing how echo can be used to display the value of your own variable:

```
$ mygreeting='Hello, what would you like to do now?'
$ echo $mygreeting
Hello, what would you like to do now?
$
```

The shell parses the preceding command and replaces $mygreeting with its value before running a process to execute echo.

You can enclose the parameter or variable name in braces:

```
$ mygreeting='Hello, what would you like to do now?'
$ echo ${mygreeting}
Hello, what would you like to do now?
$
```

The preceding command doesn't need the braces; they have no effect on its behavior. However, braces are required if the parameter is more than a single digit or if the name is followed by characters that are not supposed to be part of the name. The following examples demonstrate this:

```
$ var=123
$ echo $var
123
```

```
$ echo $varabc
$
```

This produces no output because there is no variable named `varabc`. In contrast

```
$ echo ${var}abc
123abc
$ newvar=${var}_new
$ echo ${newvar}
abc_new
```

produces output because the actual variable name is enclosed in braces so that `bash` can find it. You'll see that being able to append characters to the expansion of a variable or parameter is very useful. (See "Control Flow in the Shell" on page 19 for an example.)

File Globs

All shells have the ability to parse patterns that represent sets of filenames. These patterns are called *file globs*, *globs*, or *wildcard expressions*. The shell will expand a file glob into the list of filenames of existing files that match that glob. A string is called a glob if it contains one of the shell special characters `?`, `*`, or `[`. Each of these has a specific purpose. A question mark (`?`) not enclosed in square brackets (`[ ]`) matches any single character except the forward slash (`/`). The asterisk (`*`) matches any string not containing a `/`, including an empty string. Neither of these two wildcard characters match a leading dot (`.`). For example, if the current directory contains the files *file_io.c*, *utils.c*, *finding.c*, *.config*, and *users.o*, then

```
$ ls f*.c
```

lists the files *file_io.c* and *finding.c* because those names match that glob.

The command

```
$ rm *.o
```

removes any file ending in *.o*, which is *users.o*, and the command

```
$ gcc -c f*.c
```

runs `gcc -c` on every file in the current working directory whose name starts with *f* and ends in *.c*.

This command

```
$ ls *
```

doesn't display *.config* because it starts with a dot.

Square brackets enclosing a list of characters represent any single character in that list, unless the first character is an exclamation mark (!), in which case they match any character *not* in the list:

[abc] Matches any of a, b, and c

[!abc] Matches any character other than a, b, and c

[[] Matches [

[]] Matches]

[abc!] Matches a, b, c, and !

You can use a hyphen to represent ranges of characters inside square brackets as well:

[a-z] Matches any lowercase letter

[!a-z] Matches any character other than a lowercase letter

[A-Za-z0-9] Matches all letters and digits

[-a] Matches - and a

Be very careful when using file globs, especially with dangerous commands such as rm that are not reversible, because they may represent files that you didn't think they did. One disastrous example would be

```
$ rm -r .*
```

which a naive user might think removes the "hidden" files in the given directory and their descendants. But the pattern .* matches .. because the second period is matched by *, implying that the command will recursively remove everything in .., the parent directory.

I've only scratched the surface of how to use file globs. To learn more about them, enter

```
$ man 7 glob
```

to display their man page.

Command Substitution

Most shells have a feature called *command substitution*, which allows the output of a command to replace the actual command. There are two forms. The original syntax is

```
`command`
```

and the newer syntax is:

```
$(command)
```

These two methods have some subtle differences, and it's best to use the $(*command*) method (you will probably see the first method used in legacy bash

scripts). To illustrate how they work, suppose that *filelist* contains the names of three files, one per line: *file1*, *file2*, and *file3*. Consider the command:

```
$ wc $(cat filelist)
```

To execute this command, `bash` runs `cat filelist`, which outputs the contents of the file named *filelist*, converting the output into a space-separated list of its lines on the command line. Since *filelist* contains the lines

```
file1
file2
file3
```

`bash` replaces the text `$(cat filelist)` with the text `file1 file2 file3` on the command line, so that the actual command that it runs is:

```
$ wc file1 file2 file3
```

The result is that it runs `wc` on each of these files successively.

A second example is:

```
$ head -1 $(find repo -name "*.c" 2>/dev/null)
```

The `find` command is complex, but this use of it isn't. It searches the entire directory hierarchy rooted at *repo* for every file whose name matches the file glob `*.c`, which would match all *.c* files.

Because `find` can generate many error messages, the error output is thrown away by redirecting it to */dev/null*. The remaining lines that `find` outputs replace the command itself, separated by spaces instead of newlines. These lines will be the pathnames of all *.c* files that it finds. The `head -1` command then prints the first line of every such *.c* file.

Control Flow in the Shell

All shells are designed as programming languages with the usual palette of control flow keywords and operators. The syntax varies from one shell to another. Here I'll discuss `bash` specifically.

The most useful and important control flow constructs to learn are simple branching using `if...then...else` instructions, and looping using `for` loops. `bash` also has `while` loops, `until` loops, `case` statements, and `select` statements, but to keep this short, I don't describe them here. You can read about them in the `Compound Commands` section of the `bash` man page.

There are two types of `for` loops: an iterative one like the one commonly found in high-level languages, and a list-based one, called a *foreach* loop, which isn't present in some modern languages, most importantly, C and C++ before C++11. Their syntax is slightly different and semicolons play a critical role. The following shows the typical form of the list-based `for` loop:

```
for name in list ; do list-of-commands ; done
```

The semicolons must be present if the command is on a single line. You can also replace any of them with a newline, as follows:

```
for name in list
    do
    list-of-commands
    done
```

This example just prints the numbers 1, 2, and 3, one per line

```
$ for i in 1 2 3 ; echo $i ; done
1
2
3
```

while this second example creates a file named *nums1-200* that contains the numbers 1 to 200, one per line:

```
$ for i in $(seq 1 200) ; do echo $i ; done > nums1-200
```

It should convince you that the list-based `for` loop, combined with command substitution, can be powerful.

This third example copies each file in the current directory whose name ends in *.o* to one whose name is appended with a `_bkup` extension:

```
$ for file in *.o ; do cp  $file ${file}_bkup ; done
```

It shows how using file globs can save you lots of time, and it also introduces the `cp` command (see "Creating, Removing, and Copying Files and Links" on page 38 for more details). Notice too that braces are used to enclose the variable named `file` in the third example, because `file` is a variable name but `file_bkup` is not. The braces are needed so that `bash` can expand `$file` rather than trying to expand `$file_bkup`.

The second form of the `for` loop is like the standard C iterative `for` loop except that double parentheses are needed:

```
$ for (( i = 0; i <= 10; i++ )) ; do echo $i ; done
1
2
--snip--
10
```

This prints the numbers 1 through 10, one per line, whereas in

```
$ for (( i = 0; i <= 10; i++ )) ; do echo -n -e "$i\t" ; done ; echo
1    2    3    4    5    6    7    8    9    10
```

the numbers 1 through 10 are printed on one tab-separated line. The `-n` option tells `echo` to suppress the newline, and the `-e` option turns on interpretation of backslashed characters such as `\t` (the tab character). The `echo` after the keyword `done` adds a newline so that the prompt appears on a new line.

The branching construct in Bourne-like shells such as `bash` works as follows:

```
if list-of-commands
then
    true-branch-list-of-commands
[else
    false-branch-list-of-commands
]
fi
```

The square brackets indicate that the `else` part is optional; they aren't part of the command. The `then` part is executed if the last command in the *`list-of-commands`* is successful—in other words, if its exit status is zero. If there is an `else` part, it's executed if the last command's exit status is nonzero. For example, the following command

```
$ if ls /opt/info &> /dev/null
then
    echo /opt/info exists
else
    echo ls failed
fi
```

prints `/opt/info exists` if */opt/info* exists and `ls failed` if it doesn't.

The `if` construct was originally designed to require a list of commands as the condition to be evaluated. To create a branch based on a conditional expression instead, you had to write the `test` command after the `if` keyword. The `test` command was a separate program created specifically for this purpose. But `bash` also has a builtin command of the same name now. Its options are slightly different from the `test` program. To determine which one you get when you enter `test`, enter this command:

```
$ which test
```

The output will be the pathname to the executable file for the `test` program or will indicate that it's a shell builtin. To override the default, you can enter the full pathname, such as `/usr/bin/test` to use the `test` program, or `builtin test` to use the shell builtin version of `test`. In either case, the `test` command is followed by a conditional expression. When you run it, its return value is the truth value of the expression: either `true` or `false`. As an example, consider the following `bash` code:

```
$ if test -x myfile
then
    echo myfile is executable
else
    echo cannot run myfile
fi
```

The first line is an example of the `test` command with the `-x` operator, which checks whether a file exists and you're able to execute it. If the test is true, it prints a message that the file is executable; otherwise, it prints that the file can't be run.

Because there are slight differences between the two versions of this command and you may not know which one you're running, I'll show you an alternative syntax for the `bash` builtin `test` command. Instead of writing

```
test expression
```

you can write

```
[ expression ]
```

in which the square brackets are part of what you enter. The preceding example is therefore equivalent to this:

```
$ if [ -x myfile ]
then
    echo myfile is executable
else
    echo cannot run myfile
fi
```

To confound things further, `bash` also has a `[[ ]]` operator that has a more extensive set of expressions. Consult the man pages for further information about it.

The kinds of conditional expressions that you can form are extensive. They include:

- File tests
- Numerical comparisons
- String comparisons
- Tests of the state of variables
- Negation, conjunction (AND-ing), and disjunction (OR-ing) of any of them

File tests include tests of whether a file is readable, is writeable, is executable, is a directory, is a regular file, and so on. There are more than 20 such tests. Here are a few examples:

`test -x myfile` True if `myfile` exists and is executable by you

`test -L myfile` True if `myfile` exists and is a symbolic link

`test -s myfile` True if `myfile` exists and is not empty

`test file1 -nt file2` True if `file1` is newer than `file2`

String tests use the operators `=` and `!=`. If you use the `[[ ]]` syntax, you can also use `<` and `>` for string comparisons. The comparisons are based on

the value of your LC_COLLATE environment variable, which determines the lexicographical ordering of characters.

LEXICOGRAPHICAL ORDER

Lexicographical order is a generalization of alphabetical order. Most languages have a set of rules that define which characters precede others in their alphabets, which we call their alphabetical order. Lexicographical order adds rules for the ordering of punctuation and other nonalphabetic symbols so that an entire character set can be ordered.

To illustrate the importance of lexicographical ordering, consider the conditional expression `"Elephant" < "ant"`. The expression is true if the ordering sorts uppercase before lowercase and it is false otherwise. LC_COLLATE is one of several environment variables that are collectively known as your *locale*. Locales are introduced briefly in Chapter 1 of the book, and in more depth in Chapter 3.

The best way to learn about the various conditional expressions is to enter the command `help test` and study the full list. Here are the expressions you'll probably find most useful:

`-a myfile` True if `myfile` exists

`-d myfile` True if `myfile` is a directory

`-e myfile` True if `myfile` exists

`-r myfile` True if `myfile` is readable by you

`-s myfile` True if `myfile` exists and is not empty

`-w myfile` True if `myfile` is writable by you

`-x myfile` True if `myfile` is executable by you

`-N myfile` True if `myfile` has been modified since it was last read

`-z string` True if *string* is empty

`string1 = string2` True if the two strings are equal

`string1 != string2` True if the two strings are not equal

`! expr` True if *expr* is false

`expr1 -a expr2` True if both *expr1* AND *expr2* are true

`expr1 -o expr2` True if either *expr1* OR *expr2* is true

`arg1 op arg2` Where *arg1* and *arg2* are numeric and *op* is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, or `-ge`

Shell Scripts

If you find yourself repeating a particular sequence of commands frequently, you can make your time more productive by putting that sequence of commands into a file for later execution. We call a file containing a sequence of

commands written in a shell language a *shell script*. If the commands are written in `bash`, it's called a `bash` script; if it's written in the C shell language, it's called a `csh` script; and so on.

Once you've created a shell script—and assuming that you've taken a few steps to make it an executable file, which I'll explain shortly—to run it, you just enter its name on the command line. For example, if you've written a shell script named *myfirstscript* in the current working directory, you would enter

```
$ ./myfirstscript
```

to run its commands.

Although most of the shells have very similar syntax rules, there are variations among them, and what's true about the `bash` shell is not necessarily true about the Korn or C shell. Therefore, to be clear, everything in this section is specifically about `bash`, although much of it may be true of other shells.

Here's a simple example that illustrates two key ideas:

script1.sh

```
#!/bin/bash
# This script does nothing but print "Hello world; code responsibly!"
# Its purpose is to show what the first line looks like and
# to introduce comments.
#
# Written by Stewart Weiss, Jan. 1, 2000

echo "Hello world; code responsibly!"
```

The first line tells the shell to run the interpreter `/bin/bash`, using the rest of the file as its input. In effect it says, "I am a `bash` script." Although that line isn't strictly necessary, if you want to be able to run scripts just by entering their names, you need it. Its form is:

```
#! pathname-of-interpreter
```

In this case, the interpreter program is `/bin/bash`. If you want to write a Perl script, the first line would be:

```
#!/usr/bin/perl
```

That two-character sequence, `#!`, is known as the *shebang* symbol in the Unix world.

The next thing to note is the `#` comment delimiter; `bash` ignores anything after it on the same line (except when the second character after it is `!`). Most shells use this comment delimiter. Your scripts should always contain comments; scripts are programs, and they should be documented for all the same reasons that programs should.

In order to be able to run a script by entering its name, you also need to make that file *executable*, which means adding execute permission to the

file's permission mode. Use the `chmod` command to alter the permissions on any file that you own. For example:

```
$ chmod +x filename
```

This makes *filename* executable by everyone.

You can pass arguments to scripts on the command line, to be used inside the script, which usually makes the script more versatile. This is where certain `bash` parameters come into play. Parameters denoted by numbers other than 0 are called *positional parameters*. Thus, 1, 2, 3, and so on are positional parameters. When you enter a command, `bash` stores the successive words from the command line in successive positional parameters (1, 2, 3) up to the number of distinct words it finds on the command line. (If you enclose a string in single quotes, it will treat that entire quoted string as a single word even if there are spaces in it.)

The very first word on the command line, which is the program to run, is stored in parameter 0. Thus, in a script you can access the command line arguments with the notation `$1`, `$2`, `$3`, and so on, and the program name with `$0`. If you enter more than nine arguments, you need braces to enclose the parameters whose numbers are more than one digit, such as `${12}`.

Two particular parameters, `$#` and `$*`, are very useful in scripts. The number of arguments on the command line is stored in `$#`, and the entire set of words after the command is stored as a list of words in `$*`. The following script, which I'll store in an executable file named *script2.sh*, demonstrates how these two parameters can be used:

script2.sh

```
#!/bin/bash
#!/bin/bash
# This script shows the use of $#, $0, and $*.
# Given one or more names on the command line, it creates files with
# those names with suffix .csv, and puts two lines into each file.
# The first line is the title given to the file and the second is a
# heading.
#
# Written by Stewart Weiss, June 13, 2025
if [ $# -lt 2 ]                          # If it has no arguments
then
    echo "Usage: $0  list-of-filenames"  # Print usage message.
    exit                                 # Exit the script.
fi
for i in $*                              # For each argument
do
    echo -e "Title:  $i"  >| $i.csv      # Create file and put title in.
    echo -e "Name\tValue" >> $i.csv      # Append a heading.
done
```

It checks that `$#` is at least 2. If it's not, there are no words on the line except the script name. In this case, it uses `echo` to print a message about how to use the script (this is another useful application of `echo`). Otherwise, using a

list-based `for` loop, for each argument `i`, it writes two lines into a file named *i.csv*. Note that `i` is a variable and it's value is displayed using `$i`. This is not a positional parameter! `bash` expands `$*` into a list of words, and the `for` loop assigns each word to `i` for each iteration of its body.

If we run the script in an empty directory, providing it some arguments, and then enter the `ls` command, we see that it created a CSV file for each argument:

```
$ ./script2.sh f1 f2 f3
$ ls
f1.csv   f2.csv   f3.csv
$
```

If you open the files and look at their contents, you'll see that they have the two lines that the script wrote to them. Soon we'll see ways to open files.

We now run *script2.sh* without any arguments:

```
$ ./script2.sh
Usage: ./script2.sh  list-of-filenames
$
```

The script detects that there are no arguments and lets me know this. It also lets me how to use it correctly. All programs and all scripts should do this.

For more information about scripting, read the man page of the shell you are using.

Shell Behavior, Shell Variables, and the Environment

Shell variables were introduced and described earlier in this chapter in "Parameters and Variables" on page 15. Many of these variables affect the behavior of the shell, such as `IFS`, which stores a list of characters that `bash` treats as whitespace, and `PS1`, which defines the prompt that `bash` displays for you.

Some of `bash`'s variables are part of the environment when `bash` starts running. They're made available to `bash` when it starts. For example, the environment variable named `HOME`, which has the absolute pathname of the user's home directory, is also a `bash` variable. Another environment variable named `PATH` is a colon-separated list of pathnames of directories that the shell searches to find command names that we enter. This `PATH` variable is also accessible in `bash`.

An easy way to understand the difference between environment variables and shell variables is by analogy to the difference between global variables and local variables in a program. Global variables are defined outside of the program, in file scope. They are like environment variables, which exist outside of the shell. In contrast, locals are defined inside the scope of the program; they're analogous to shell variables. Just as both global and local variables are accessible to the program, environment and shell variables are accessible to `bash`. Like globals and locals, they're stored in different places and receive initial values in different ways.

You can see the full list of bash's shell variables by entering the set command. You can see the value of any bash variable, whether it's from the environment or not, with the echo command, as in:

```
$ echo My current working directory is $PWD
My current working directory is /home/stewart/demos
$ echo $HOSTNAME is running $OSTYPE on machine type $MACHTYPE
harpo is running linux-gnu on machine type x86_64-pc-linux-gnu
$
```

The POSIX.1-2024 standard (available from *https://pubs.opengroup.org/onlinepubs/9799919799/*) lists 16 different environment variables that can influence the shell's behavior.

When you first log in, the kernel initializes your environment and starts up a new process for your login shell, passing it a reference to that environment. It gets the initial values of your environment variables from a few different sources, including various system files such as */etc/environment*, */etc/profile*, and */etc/bash.bashrc*, as well as user-specific files in your home directory, such as *.bash_profile* and *.bashrc*.

Shell variables that do not come from the environment, such as OSTYPE, MACHTYPE, and HOSTNAME, get their initial values from other sources, such as configuration files.

Once you've logged in and you're working within a shell, when you run a non-builtin command or a program, a new process is created and it inherits the environment of the shell from which you ran it. More generally, whenever a new process is created, it inherits a copy of its parent's environment, even if the parent is not a shell. This is how programs know what their current working directories are, for example, or which users are their owners, or in which time zone they're running. The environment is a key element in making things work. It's like a set of global variables available to all processes.

You're free to customize the environment of a shell by defining new variables or redefining the values of existing variables. The syntax is shell dependent. In the Bourne shell and bash, variables that you define, which we'll call *user variables*, are not automatically placed into the environment. To place them into the environment, you have to *export* them using the export command. Exporting a variable essentially makes that variable available to all processes that you run afterward, including any subshells. For example, if you frequently visit a directory that has a long pathname and you want to access it easily from anywhere in the directory hierarchy, you could create a variable that stores the pathname, such as the following:

```
$ export LECTURES=/home/stewart/teaching/unixclass/lecturenotes/
```

This command puts LECTURES into the environment with the value /home/stewart/teaching/unixclass/lecturenotes/ so that it can be inherited by future commands and processes. This pair of commands does the same:

```
$ LECTURES=/home/stewart/teaching/unixclass/lecturenotes/ ; export LECTURES
```

The semicolon ends the first command and a newline ends the second. Now you can navigate to that directory by entering `cd $LECTURES`.

Theconvention is to use uppercase names for environment variables and lowercase names for variables that you define and do not export. The following session demonstrates what happens both when you don't export and when you do:

```
$ WORKDIR=/home/stewart/scratch
$ echo $WORKDIR
/home/stewart/scratch
$ bash      # Start a new shell.
$ echo The value of WORKDIR is $WORKDIR
The value of WORKDIR is
$ exit      # Exit new shell to previous one.
$ export WORKDIR
$ bash      # Start a new shell again.
$ echo The value of WORKDIR is $WORKDIR
The value of WORKDIR is /home/stewart/scratch
```

We start by assigning a value to `WORKDIR` without exporting it. It's a `bash` user variable but not in the environment. When we start a subshell by entering `bash`, it doesn't get a copy of this variable, so the `echo` command outputs an empty string for `$WORKDIR`. We exit the subshell and, this time, export the variable `WORKDIR`. We start a subshell again and run `echo`, passing `$WORKDIR`. Now it sees it and can output its value.

The History Mechanism

After a while of working in the command line, you'll find that you wish you had an easy way to enter the same command that you ran some time earlier, either exactly as you did before, or perhaps with a different set of arguments. In short, you'll wish your shell had a means of remembering what commands you typed.

Some shells, including `bash`, provide this facility. They can save a list of the commands you've previously entered into the shell, for later recall. This feature is commonly known as the *history mechanism*. The history mechanism is an optional feature that's usually enabled by default. One way to see if it's enabled is:

```
$ set -o | grep history
history    on
```

Most systems have it enabled by default. If your output says that it's off, you'll need to edit your *.bashrc* file to enable it by including the line `set -o history`.

Roughly speaking, `bash` saves the commands that you enter in an internal list that you can think of as an array of strings. This internal list is its *history list*. When you exit the shell, it writes this list into a file, unless you disabled saving the list. The default pathname of this file is *~/.bash_history*. Whenever you start a new shell, it initializes its history list from the contents

of the history file. I won't discuss the complex issues such as how many lines the file has, how saving works when the file isn't empty, and so on; the `bash` man page explains the details.

Let's see how you can recall previous commands. For starters, if you want to see the entire history, which by default is usually about 500 commands long, enter **`history`**. Each line of output will have a number followed by a command that you executed, such as:

```
50  cd  demodir
51  make demo2
52  ls .
53  date
54  echo "testing echo"
...
```

The number is called the *command number*. You can re-execute any command in this list by entering an exclamation mark (`!`) at the prompt, followed by its number:

```
$ !54
echo "testing echo"
testing echo
```

First `bash` displays the command, and then it executes it.

Commands in the list are called *events*, and the notation `!`*n*, where *n* is a specific command number, is called an *event designator* since it designates a specific event. Following is a list of some useful event designators:

`!!` Previous command

`!`*n* *n*th command line in the history list

`!-`*n* *n*th most recent command in the list

`!`*str* Most recent command that started with the string *str*

`!?`*str*`?` Most recent command that contained the string *str*

Given the preceding sample history output, if I want to re-execute the `make` command, I could enter either of the following:

```
$ !51
make demo2
$ !make
make demo2
```

Even easier, if you press the up arrow key on the keyboard in `bash`, it displays the preceding command. Press it again and you get the one before that, and so on. If the last command I executed had command number 56, then pressing the up arrow five times displays the `make demo2` command, and I can just press ENTER to re-execute that command.

There's much more to learn about the history mechanism, but I don't include it here. Not only can we re-execute commands, but we can also pick

out arguments from them to reuse in the current command or repeat a previous command, substituting any number of characters in it for others. To give you a sense of it, assuming once again that the history is the one listed earlier in this section, to run `make` to build *demo3* instead of *demo2* (assuming *demo3.c* exists in my directory), I can enter

```
$ !make:s/2/3/
make demo3
```

which replaces the first occurrence of `2` in the matching command with `3`.

I encourage you to read more about this in the `bash` man page.

Types of Commands

Commands are not always executable programs stored in files. In general, a command might be any of the following types:

- A *binary executable file*, such as `ls`, which is the file */bin/ls*
- A *shell builtin* (defined in Chapter 1 of the book, in the section "Shells" on page 12) such as `alias`, `cd`, and `exit`
- An *alias* (alternate name) for another command, defined either by you or by an administrator of the computer using the `alias` builtin command, as in `alias rm='rm -i'`, which replaces the dangerous `rm` command with a version of it that always asks before deleting anything
- A *script*, such as a shell script or a script in another scripting language such as Ruby, JavaScript, and Perl

Sometimes the same command name can denote more than one of these. For example, commands like `pwd` and `echo` are both shell builtins and executable files. You can determine the type of the command using the `type` `bash` builtin, which takes one or more command names:

```
$ type who ls cd backup
who is /usr/bin/who
ls is aliased to `ls -F --color=tty'
cd is a shell builtin
backup is /home/stewart/bin/backup
```

This tells you whether it's a file, an alias, or a builtin.

If `type` reports that a command is a file, you can use the `file` command to determine whether it's a binary file or something else:

```
$ file /home/stewart/bin/backup
/home/stewart/bin/backup: Bourne-Again shell script, ASCII text executable
```

The `file` command attempts to detect exactly the type of file. In this case, it reads the first line and finds that it started with the `#!/bin/bash` command.

File Permissions

Before we explore the various commands for working with files, you need to understand how file permissions work. In this context, the word *file* refers to all file types, including directories.

Unix has a very simple but powerful method of protecting files. To provide a way for users to control access to their files, the inventors of Unix devised a rather elegant and simple access control system. Every file has three modes of access: read, write, and execute.

Read access The ability to view file contents. For directories, this is the ability to view the contents of the directory using the `ls` command.

Write access The ability to change file contents or certain file attributes. For directories, this implies the ability to create new links in the directory, to rename files in the directory, or to remove links in the directory. This might sound counterintuitive. The ability to delete a file from a directory does not depend on whether one has write privileges for the file, but on whether one has write privileges for the directory.

Execute access The ability to execute the file. For a directory, execute access is the ability to `cd` into the directory and, as a result, the ability to run programs contained in the directory and run programs that need to access the attributes or contents of files within that directory. In short, without execute access on a directory, there is little you can do with the files contained in it.

For each file, three different classes of users are defined, and each class can have a different set of allowed modes of access:

- A file has an owner, called its *user*. The *user class* contains the owner and no one else. *User access* is the set of access rights given to this class.
- A file is also associated with a group of users, called its *group*. The owner of a file can assign any of the groups to which the owner belongs to that file (with the `chgrp` command). Users in that group other than the owner belong to the *group class* and have *group access* rights to the file.
- Everyone who is neither the user nor a member of the file's group is in the class known as the *others class* and has *others access* rights to the file.

In short, with respect to a single file, every user in the system belongs to exactly one class—either the user class, the group class, or the others class. Their permissions for this file are determined by the class they're in, namely one of user access, group access, or others access. I remember this with the acronym *UGO*.

For each of the three classes of users, 3 protection bits define the read, write, and execute privileges afforded to members of the class. For each

class, if a specific protection bit is set, then for anyone in that class, the particular type of access is permitted. Thus, there are 3 bits called the **r**ead, **w**rite, and e**x**ecute bits for the user (`u`), for the group (`g`), and for others (`o`), for a total of 9 bits. These bits, which are called *mode* or *permission bits*, are usually expressed in one of two forms: as an octal number, or as a string of nine characters.

The nine-character permission string is best understood as three groups of three characters, where the characters representing the high-order bits are to the left:

```
r w x        r w x        r w x
user bits    group bits   others bits
```

In a permission string, we usually represent an enabled bit with the letter associated with it, and a disabled bit with a hyphen. Many commands use this convention as well. Some examples are shown here:

`rwxrw-r--` The user (the owner) has read, write, and execute permission, the group has read and write but not execute, and others have only read permission.

`r-xr-xr-x` Everyone has read and execute permission but not write permission.

`rwxr-xr--` The user has read, write, and execute permission, the group has read and execute permission, and others have only read access.

`rw-r-----` The user has read and write access, the group has read access only, and no one else can do anything with the file.

The mode string can also be represented as a three-digit octal number by treating each group of 3 bits as a single octal digit. We can use the C ternary operator, `?:`, to express the conversion of permission letters to an octal value:

```
octal_value = r?4:0 + w?2:0 + x?1:0
```

In other words, the read bit is 4, the write bit is 2, and the execute bit is 1. We add the values of the enabled bits to get a single number in the range `0` to `7`, which results in the following table of values for each group of 3 bits:

```
rwx   rw-   r-x   r--   -wx   -w-   --x   ---
 7     6     5     4     3     2     1     0
```

For example, to compute the octal number for the mode `rwxrw-r--`, break it into the three substrings `rwx`, `rw-`, and `r--`, and then use the above conversions on each substring: The user sum (`rwx`) is 7, the group sum (`rw-`) is 6, and the others sum (`r--`) is 4, resulting in octal number 764.

In addition to the mode bits, a file's permission string is often displayed along with a single character code that represents the file's type. This character is usually written to the left of the mode string and can be one of the following:

- `-` A regular file
- `d` A directory
- `b` A buffered special file
- `c` A character special file
- `l` A symbolic link
- `p` A pipe
- `s` A socket

The ones you're most likely to see are directories, regular files, and symbolic links.

Viewing and Modifying File Attributes

Sooner or later, we all try to run a command that fails because we don't have the permission to access some file in the way we tried. Knowing how to view and change file permissions can help us avoid this problem.

To see several attributes of a file, including its mode, we can use the `-l` option (for *long* listing) of the `ls` command. When `ls` is given this option, for each file, it prints the mode string, the number of links to the file, the username of the owner, the group name of the file's group, the number of bytes in the file, the last modification time, and the file's name in the given directory. For example, we can look at the attributes of the file named *meeting_notes* in the current working directory by entering the following:

```
$ ls -l meeting_notes
-rw-r--r-- 1 stewart faculty 3304 Sep 22 13:05 meeting_notes
```

The output tells us that this is a regular file because its type is `-`, and it can be read and written by its owner (`rw-`). Anyone in the `faculty` group (`r--`) can only read it, and anyone else (`r--`) can only read it. The `ls` command has several other options for controlling the information that it displays. Read its man page for more details.

We can use the `stat` command to display even more attribute information. We give `stat` one or more files, as shown here:

```
$ stat lib bin
  File: lib
  Size: 4096       Blocks: 8          IO Block: 4096   directory
Device: 813h/2067d Inode: 6684786     Links: 14
Access: (0750/drwxr-x---)  Uid: (  500/ stewart)   Gid: (  500/ stewart)
Access: 2025-06-13 16:12:01.096641180 -0400
Modify: 2018-07-25 14:16:29.290584068 -0400
```

```
Change: 2018-07-25 14:16:29.290584068 -0400
 Birth: 2018-07-25 14:16:28.814596709 -0400
  File: bin
  Size: 4096       Blocks: 8          IO Block: 4096   directory
Device: 813h/2067d Inode: 3670282     Links: 2
Access: (0750/drwxr-x---)  Uid: (  500/ stewart)   Gid: (  500/ stewart)
Access: 2025-06-13 16:12:00.772649653 -0400
Modify: 2025-04-08 09:53:24.724151675 -0400
Change: 2025-04-08 09:53:24.724151675 -0400
 Birth: 2018-07-25 12:06:50.619102451 -0400
```

This command outputs some information about files that I haven't yet described. In Chapter 6 of the book, we'll learn what all of its output means, and we'll develop an implementation of it.

For ordinary text files, we can get the counts of the bytes, words, and lines in one or more files with the `wc` command. We can give it one or more filenames as arguments, as shown here:

```
$ wc README.md README.txt
  72  401 2559 README.md
  65  382 2417 README.txt
 137  783 4976 total
```

It outputs counts of the lines, words, and bytes, in that order, followed by the filename. The output shows that there are 72 lines, 401 words, and 2,559 bytes in the file named *README.md*. Other options provide different information.

The most important commands that change the attributes of a file are:

`chmod` ***`mode [,mode] ... files`*** Changes the permissions on *`files`*

`chown` ***`owner files`*** Changes the ownership of *`files`*

`chgrp` ***`group files`*** Changes the group ownership of *`files`*

`touch` ***`files`*** Updates timestamps of *`files`* or create empty ones if they don't exist

Of these, `chmod` is the one you'll probably use most frequently. The command's name is a mnemonic: ***ch****ange* ***mod****e*. Most people pronounce it as "C H mod." The `chmod` command has a complex set of options and accepts permissions written as letters or as octal numbers.

In the simplest case, a mode is of the form

```
user-designation operator type-of-permission
```

where *`user-designation`* is one or more of the letters `u`, `g`, `o`, and `a`, *`operator`* is one of `+`, `-`, and `=`, and *`type-of-permission`* is any nonempty subset of `r`, `w`, and `x`. No spaces are allowed between these three parts! The letter `a` is short for *all users* and means all of `ugo`. The `+` operator adds the given permissions to

the designated users, and the - operator removes the given permissions. The = operator sets the permissions to exactly the ones specified, removing any others that might have existed. Examples follow, using the single file named *myfile* to illustrate:

`chmod u+rw myfile` Adds read and write permission for the owner of `myfile`

`chmod g+rx myfile` Adds read and execute permission for the group of `myfile`

`chmod go-w myfile` Removes write permission for everyone in the group and others for `myfile`, so that only the owner can modify `myfile`

`chmod u=rwx myfile` Sets the mode to `rwx------` for `myfile`

`chmod u=rw,og=r myfile` Sets read and write permission for the owner, read-only for everyone else

`chmod a-x myfile` Removes execute permission on `myfile` for all users

`chmod +x myfile` Absent any user designation, applies to all, so this adds execute permission to `myfile` for everyone

`chmod 755 myfile` Gives `myfile` permission `rwxr-xr-x`

`chmod 640 myfile` Gives `myfile` permission `r-xr--r--`

I haven't described all the ways we can use the `chmod` command. Read its man page to learn more about it.

The `touch` command is pretty convenient for creating empty files and for updating the time of last modification of one or more files, which is convenient when you're using the `make` command. By `touch`ing one or more files, you can force `make` to recompile a program that depends on them. The `make` command is a powerful command for automatically updating a program or some other file that depends upon other files. I don't cover `make` here, but I wrote a hands-on tutorial for `make` that you can download from the book's GitHub repository at *https://github.com/stewartweiss/Make-Tutorial*.

Here is how you can create a bunch of empty files using `touch`:

```
$ ls    # List the directory to see files don't exist yet.
$ touch f1 f2 f3 f4 f5  # Create the files.
$ ls -l
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f1
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f2
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f3
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f4
-rw-r--r-- 1 stewart faculty 0 Jan  9 12:40 f5
```

This shows that `touch` created five empty files within the same minute.

Working with Directories

You don't need to know many commands in order to do most directory-related tasks in Unix. The most common tasks fall into these categories:

Navigation Changing or identifying the current working directory

Display Showing the contents of directories

Creation Creating new directories

Restructuring Renaming, moving, and removing directories

Content modification Changing the content of directories

Following is a list of the essential commands with explanations of their simplest use cases. I don't describe the various options or details of each command's usage; for that you should read their respective man pages.

`pwd` Prints the absolute pathname of the working directory. The *p* in `pwd` stands for *print*, but it does not print on a printer. In Unix, *printing* means displaying on the screen. (That's why the C instruction `printf()` sends output to the display device, not the printer.)

`cd [`*`dir`*`]` Changes the working directory to *dir*, if *dir* is given, and to the home directory if no argument is given. The `cd` command (think *change directory*) changes your current directory. You give it the name of a directory, either as an absolute pathname or as a relative pathname.

`ls [`*`dir1`*`] [`*`dir2`*`] ...` Displays the contents of *dir1*, *dir2*, and so on if they're supplied; otherwise, display the contents of the current working directory.

`mkdir `*`dir1`*` [`*`dir2`*`] ...` Creates new directories named *dir1* (and *dir2*, and so on). If *dir1* (and *dir2*) are pathnames without slashes in them, they are created in the current working directory; otherwise, they're considered to be pathnames, and they're created in the directories specified by their paths. The `mkdir` command is the only one that creates a directory. If the name you choose already exists, `mkdir` will fail.

`rmdir `*`dir1`*` [`*`dir2`*`] ...` Removes the *empty* directory *dir1* (and *dir2*, and so on). You cannot use `rmdir` to remove a nonempty directory (see the next command).

`rm -r `*`dir1`*` [`*`dir2`*`] ...` Removes all entries in all directories (*dir1*, *dir2*, and so on) and removes the directories themselves. This command deletes one or more directories and their contents, but it's not reversible, so be careful. There is no "trash can" associated with `rm` from which you can restore deleted files. The `rm` command, without the `-r` option, is described in the next section, "Working with Files." Commands that delete files, such as `rm`, and commands that create or rename files, actually modify directories, not files! They're also covered in the next section.

`ln `*`file1 file2` Makes a new directory entry named *file2* for the file named *file1*. It isn't a copy of *file1*; it's another name for it. In "Creating, Removing, and Copying Files and Links" on page 38, I'll explain more about it.

`mv` *`files`* Renames files and/or directories. It's short for "move" because it can move a file (or directory) into another directory as well as rename a file within the same directory. It has several different synopses, which I'll also explain more about in "Creating, Removing, and Copying Files and Links."

Changing directories and being in a directory are imprecise phrases. When you `cd` to a directory named *dir*, you may think of yourself as being *in dir*, but that's not accurate. What is true is that the *dir* directory is now your current working directory, and every process you run from the shell in which you changed the directory, including the shell process, will use *dir* by default when it's trying to resolve relative pathnames.

Working with Files

I classify commands for working with files as those that

- Display file contents but do not modify them
- View and possibly modify file attributes
- View and modify file contents in one way or another
- Create, remove, rename, or copy files

We saw commands for viewing and modifying file attributes in "Viewing and Modifying File Attributes" on page 33. Commands that can modify files are called *editors*. Editors are a separate topic, which I don't cover in this book. I'll first describe commands for viewing files, after which I'll describe those that create, remove, rename, and copy them.

Displaying Files

Here I list and summarize the basic commands for viewing the contents of files, not editing or modifying them. For all of the commands I list here, if you don't supply any file arguments, the command reads from standard input. For those commands that accept more than one file argument, the command processes its arguments in the order they appear on the command line.

`cat [`*`file`*` [`*`file`*`] ... ]` Displays all file contents

`more [`*`file`*` [`*`file`*`] ... ]` Displays all file contents a screen at a time

`less [`*`file`*` [`*`file`*`] ... ]` Displays all file contents a screen at a time with more options

`head [-`*`n`*`] [`*`file`*` [`*`file`*`] ... ]` Displays the first *n* lines of each file; the default is *n* = 10

`tail [-`*`n`*`] [`*`file`*` [`*`file`*`] ... ]` Displays the last *n* lines of each file; the default is *n* = 10

`diff` *`file1 file2`* Compares two files, line by line

`cmp` *`file1 file2`* Compares two files, byte by byte

The names of the `more` and `less` commands have an interesting story. The `more` command, which existed long before there was a `less` command, was created to allow you to read a file, or standard input, one screenful at a time, and also to jump ahead in the file by searching for patterns called *regular expressions*, using the same search operator as is used in the `vi` editor. However, it wasn't easy to go backward in the file with `more`. Navigation eventually improved in modern versions of `more`.

The `less` command was written and released by Mark Nudelman in 1985 to do more than `more`, but someone he knew suggested the name `less` (see *http://www.greenwoodsoftware.com/less/faq.html#history*). I like to think of this as a play on the adage "Less is more," often attributed to the German-born Modernist architect Ludwig Mies Van der Rohe, intended to convey that simplicity is better. More likely than not, it was so named because it lets you see less of a file as well as more of it. No matter why they named the enhanced version of `more` "`less`," `less` does more than `more`. The `less` command is the most versatile means of viewing files, and you should learn how to use it.

The `head` and `tail` commands are very useful. I often want to see just the first line of a file, or the first few lines, which is what `head` does. By combining `head` and `tail`, you can output any arbitrary contiguous sequence of lines of a file. (The `sed` filter does this more easily.)

The `diff` command comes in handy when you want to know whether two files are exactly the same, and if not, how they differ. By default, if they're identical, `diff` returns with no output. If they differ, it displays the differences. If you use the `-s` option to `diff`, it displays a message when the two files are the same, which is sometimes preferred. A similar command is `meld`, which presents the two files side-by-side in a separate pop-up window. If it isn't on your system, you can install its package. The `cmp` command is useful for comparing binary files.

Creating, Removing, and Copying Files and Links

When you create, remove, or rename a file, you're modifying the directory in which that file's name appears. Therefore, you need to have write permission on any directory in which you want to perform any of these operations, regardless of which command does it. In fact, technically all of the commands in the following list should be categorized as directory modification commands, but because they also have effects on files, I put them here. Almost all of these commands have more than one form, but I show only a few of the different forms.

`ln` *`f1 f2`* Creates a new link for nondirectory file *`f1`* named *`f2`*.

`ln` *`f1`* `[` *`f2 ...`* `]` *`dir`* Creates a new link for each nondirectory file, with its same name, in the directory *`dir`*. The last argument must be a directory. The `ln` command can be used to create new names only for files, not directories. If the new name already exists, it displays an error message. You can read about this command's many options on its man page. A simple version of `ln` is named `link`.

`rm` *`files`* Deletes the directory entries for all given *`files`*. If a file has no more names in any directories, it is permanently removed. The `rm` command is irreversible; once a file is removed, it cannot be recovered. This command does not remove directories unless the `-r` option is given to it, in which case it deletes an entire hierarchy rooted at the given directory.

`mv` *`f1 f2`* Renames *`f1`* with the *new* name *`f2`*. Note that if *`f2`* is an existing name of a nondirectory file, it will be replaced, provided that *`f1`* is not a directory. If *`f2`* is an existing directory, this is the form of `mv` that follows next. Because `mv` will silently overwrite existing files, you should use `mv -i`, which prompts you before overwriting them.

`mv` *`files dir`* Moves all files into the destination directory *`dir`*. In this form, the last argument must be an existing directory name, and all preceding arguments are moved into that directory. The preceding arguments can be existing directories, as long as moving them does not create a cycle in the directory hierarchy, such as trying to move the current directory into one of its subdirectories.

`cp` *`f1 f2`* Makes a copy of *`f1`* named *`f2`*. The two names must be different.

`cp` *`f1`* [*`f2 ...`*] *`dir`* Copies all files into the destination directory *`dir`*. The `cp` command works almost the same way as the `mv` command, except that it replicates files instead of moving them. The main difference is that `cp` does not accept directories as arguments, except as the last argument, unless you give it the `-r` option. If you give it the `-r` option, as in `cp -r`, it recursively copies the directories. Note that `cp` makes copies of files, not just their links, so if you change the copy, you are not also changing the original.

We'll go through some examples to demonstrate how these commands work.

```
$ cd branch                     # Change working dir to branch.
$ ls                            # Display current contents of branch.
project.c
$ mv project.c finalversion.c   # Rename project.c.
$ ls                            # Display branch again to see the changes
finalversion.c.
```

This shows that the `mv` command changes the link *project.c* to *finalversion.c*. We need to use `ls` to see that it does this because all we really did was change the parent directory, *branch*, by replacing the link named *project.c* with a link named *finalversion.c*.

In contrast, consider this example:

```
$ cd master
$ ls
io.c io.h main.c ui.c  ui.h utils.c  utils.h
$ mkdir headers ; mv *.h headers
$ ls
headers io.c main.c ui.c   utils.c
```

In this case, we create a new directory named *headers* in the directory *master* and use `mv` not to rename files, but to move their links into the newly created *headers* directory.

In the following example

```
$ rm hwk1.o project.o main.o
```

the `rm` command removes three file links from the current working directory. If these files have no other links in other directories, the attributes and contents are also deleted from the filesystem; otherwise, just the names are removed.

In this example

```
$ ls .                                                  # Display . files.
project.c main.c utils.c
$ mkdir /sources/myrepos/project2                       # Create new directory.
$ ln project.c main.c utils.c /sources/myrepos/project2/  # Make links there.
$ ls /sources/myrepos/project2                          # Display new directory.
project.c main.c utils.c
```

we use `ln` to create a new link for each of *project.c*, *main.c*, and *utils.c* in the */sources/myrepos/project2/* directory with their same names.

Consider this example:

```
$ cp -r main.c utils.c utils.h images ../version2
```

It copies the three files *main.c*, *utils.c*, and *utils.h*, and the directory *images*, into the directory *../version2*.

Filters in Brief

A *filter* is a program that gets its input from its standard input stream (*stdin*), transforms it, and sends the transformed input to its standard output (*stdout*). The data passes through the filter, which typically has command line options that control its behavior. A filter may also perform a *null* transformation, meaning that it makes no change at all to its input. The `cat` command is a null filter. Filters process text only, either from input files or from the output end of another command (in other words, through a *pipe*). All filters can be given optional filename arguments, in which case they take their input from the named files rather than from standard input. For example, in the command

```
$ cat first second third > combinedfile
```

`cat` reads files `first`, `second`, and `third` in that order and concatenates their contents, sending them to the standard output, which has been redirected to a file named `combinedfile`.

Following is a list of most of the filters:

awk Pattern-matching, field-oriented filter and full-fledged Turing computable programming language

cat Essentially a null filter

cut Removes sections from each line of its input stream

fold Wraps each input line to fit in a specified width

grep Filters based on regular expressions

head, tail Removes all but the top or bottom set of lines from its input stream

join Joins lines of two files on a common field

od Dumps files in octal and other formats

paste Merges lines of files

sed Line-oriented text editing filter

shuf Generates random permutations

sort Sorts using a wide range of methods

split Splits a file into pieces

tac Concatenates and print in reverse order

tr Translates or deletes characters

uniq Reports or omits repeated lines

If your time is limited and you have time to learn only one of these, the most important would be `grep` because the return on your investment will be greatest. Coming in second would be `sed`, and then `awk`. The remaining filters are easy to learn and use and are described briefly first.

In the next few sections, I'll describe some of the filters that I find most useful.

The sort Filter

The `sort` command is easy to use:

```
$ sort file
```

This sorts the text file named *file* and prints it on standard output. By default, it uses your current collating order from the environment variable `LC_COLLATE`. In many systems, the default is that all uppercase letters precede all lowercase letters. There are versions of `sort` that ignore case by default, but if yours doesn't, you can turn off case sensitivity with the `-i` option.

If you want to sort numerically, use the `-n` option, as in

```
$ sort -n numeric-data-file
```

which will sort numbers correctly. If you omit the `-n`, `9` will precede `10` because `1` precedes `9` in the collating sequence. The `sort` man page has all of the details.

The cut Filter

The cut filter is one that you may not use much, but it's easy to learn and it simplifies various tasks. It removes the same section of each line of its input, like removing columns from a CSV file. This removes everything from each line of *myfile* except the first ten characters:

```
$ cut -c1-10 myfile
```

This removes everything from each line of *myfile.csv* except fields 2 and 4:

```
$ cut -f2,4 myfile.csv
```

By default, fields are delimited by the tab character unless you change the delimiter. If *myfile.csv* is a tab separated–values file, then this deletes columns 2 and 4 from it. You can change the delimiter with the `-d` option. For example, if *myfile.csv* used commas to separate fields, then

```
$ cut -d, -f3-5 myfile.csv
```

will remove all columns except columns 3 through 5 of *myfile.csv*.

Fields are 1-based: The first field on a line is field 1, not field 0. Also, the delimiter must be a single character. Since the */etc/passwd* file uses colons (`:`) to separate fields, we pass the colon as the delimiter:

```
$ cut -f1,3 -d: /etc/passwd
```

This outputs only fields 1 and 3 of the */etc/passwd* file, which are the username and user ID fields of each entry.

Regular Expressions and grep

The single most important filter to learn is grep. It's powerful and immensely useful. To use it, we need to understand regular expressions. The regular expressions used by grep are the same as those used by sed, by the visual text editor, vi, and by awk. Therefore, taking the time to learn them is well worth it.

The simplest form of the grep command is

```
$ grep regular-expression files
```

where *regular-expression* is an expression that represents a set of zero or more strings to be matched. The syntax and interpretation of regular expressions is explained in the regex(7) man page as well as in the man page for grep itself. Entering either

```
$ man 7 regex
```

or

```
$ man grep
```

will show you all you need to know to use them. The simplest patterns are strings that do not contain regular expression operators of any kind; those match themselves. For example

```
$ grep print file1 file2 file3
```

prints each line in files *file1*, *file2*, and *file3* that contains the word *print*. It will print these lines in the order in which the files are listed on the line. If you want just a count of the matching lines, use the `-c` option; this omits the actual lines and prints just the count. If you want the nonmatching lines, use the `-v` option. This is like an inverse operation, printing all nonmatching lines. If you want to see the line numbers in addition to the actual lines, use the `-n` option. The `grep` man page describes many more options.

If you want to match a string that contains characters that have special meaning to the shell, such as whitespace, asterisks, slashes, dollar signs, and so on, it should be enclosed in single quotes:

```
$ grep 'atomic energy' file1 file2 file3
```

This will match all lines in the given files that have the exact string `atomic energy` somewhere in the line.

NOTE *The lines that `grep` finds matches for do not have to be exactly the pattern; they only have to contain the pattern.*

If you want a pattern to match an entire line, you have to bracket it with operators called *anchors*. The start-of-line anchor is the caret (`^`) and the end-of-line anchor is the dollar sign (`$`):

```
$ grep '^begin$' file1 file2 file3
```

This matches only those lines in the given files that consist of the string `begin` with no leading space and no following space.

Regular expressions can be formed with various operators. One is the asterisk (`*`), which multiplies the expression to its left zero or more times. For example, `a*` matches `a`, `aa`, and `aaa`, as well as any string containing no `a`'s.

To match multiple occurrences of a string such as `ab`, you have to enclose it with a grouping operator, `\( \)`, before the asterisk, as in

```
\(ab\)*
```

which matches zero or more sequences of `ab`. If you instead use

```
(ab)*
```

it will match strings such as `(ab)(ab)(ab)` but not `ababab` because in regular expressions, the parentheses by themselves do not perform grouping. The `*` operator is an example of a *repetition* operator. Regular expressions include other repetition operators, such as `\+`, `\?` and `\{ \}`.

The period (.) matches any character. Regular expressions also include *character classes*, which are formed by enclosing a list (or a range) in square brackets ([ ]). A character class represents a single character from that class.

Because the special characters in regular expressions typically have special meaning in the shell as well, it's a good idea to always enclose the pattern in single quotes. In particular, if you give it a regular expression using an asterisk, you should enclose the string in single quotes. Single quotes are better than double quotes. Single quotes prevent the shell from doing any interpretation of the enclosed characters, whereas when the shell sees a double-quoted string, it does a certain amount of interpretation. Until you understand what the shell will attempt to interpret inside double-quoted strings, use single quotes for enclosing grep patterns.

Examples

In the following examples, the file argument is omitted for simplicity. In this case, grep would apply the pattern against the standard input, which means if you actually enter these examples, it will wait for you to enter lines of text in the terminal and, after each line, it will either display the line if it matches, or display nothing. Terminate input with CTRL-D.

The following matches any line containing the word while followed by zero or more space characters, followed by a parenthesized expression of any kind:

```
$ grep 'while *(.*)'
```

This example matches lines that begin with a word that starts with either a lowercase or uppercase letter or underscore, followed by zero or more letters, digits, or underscores:

```
$ grep '^[a-zA-Z_][a-zA-Z0-9_]*'
```

These are precisely the set of valid C identifiers.

This pattern selects strings that have one or more digits followed by a single period, followed by exactly two digits:

```
$ grep '[0-9][0-9]*\.[0-9][0-9]\>'
```

The period must be preceded by a backslash so that grep does not treat the period as the special character that matches any character. The \> at the end tells grep to *anchor* the pattern to the end of a word. A word is a sequence of letters, underscores, and/or digits. This pattern forces grep to select only those words that end in two digits. If I omitted the \>, grep would have matched strings such as 1.234 or 1.23ab. There's a symmetric matching operator, \< that anchors a pattern to the beginning of the word.

Now take a look at this example:

```
$ grep '\/*.*\*\/'
```

Since / is a special character, if I want to match it I have to escape it with a \, as in \/. Similarly, since * is a special character in regular expressions, * is how I have to match a single literal asterisk, *. So, to match the two-character sequence /*, which is the start of a C comment, I have to write \/*, and to match /* followed by any number of characters and then followed by */, I have to write

```
\/\*.*\*\/
```

in which .* matches zero or more characters of any kind (including the period itself). This finds lines with C-style comments in them.

Regular expressions also provide a means of remembering matched expressions for reuse in the expression. This is very handy in vi and sed, which have substitution operators. The same operator used for grouping is also used for remembering matching strings. In other words, \(*pattern*\) squirrels away parts of the input line that match *pattern*.

The matching string is saved in a special variable that you can use later in an enclosing pattern or in a substitution operation. Regular expressions use variables named \1, \2, \3, and so on to store matching strings from the text.

Let's look at some examples. In this simple one

```
$ grep '\([a-z]\)\1\1\1\1'
```

the pattern matches any line that contains a sequence of five copies of the same lowercase letter, such as xxxxx or bbbbb.

Here's another example:

```
$ grep '\([1-9][0-9]\).*\1'
```

This pattern begins with a subpattern that matches any numeral consisting two digits. The next part, .*, matches any sequence of characters, including an empty sequence. The last part is \1, which stores whatever string matched the two-digit numeral. Therefore, this pattern is matched by any line that has a two-digit number that is repeated later in the line.

The command

```
$ grep '\([a-z]\)\([a-z]\)\([a-z]\)\3\2\1'
```

finds lines with palindromes of length 6. It begins by remembering three lowercase letters in succession. It then uses the saved matches in reverse order in the \3, then \2, then \1. Each variable has a copy of the single lowercase letter that was saved, so this pattern matches palindromes of length 6 such as xyzzyx.

I encourage you to read the grep man page. It has a lot more about regular expressions than I covered here. The best way to learn them is to experiment. You can open a terminal window and enter **grep** followed by a pattern. It will then wait for you to enter lines on the keyboard. Lines that match will be repeated. Lines that don't will not. Try it!

The Rest of the grep Family

Two other commands, egrep and fgrep, perform slightly different functions than grep. Although they still exist, their use is deprecated. Instead, their exact behavior is achieved by passing either -E or -F to grep. In other words, grep -E is the same as egrep, and grep -F is the same as fgrep. I'll refer to them here as egrep and fgrep for simplicity.

Think of egrep as short for extended grep. It has a simpler set of regular expressions and meta-symbols than grep, including |, ?, +, {}, and ordinary parentheses. These take the place of \|, \?, \+, \{ \}, \(, and \) in grep, respectively.

For example, you can write

```
$ egrep 'March|April|May'
```

which matches any line containing any of the month names March, April, or May, which you'd have to write with grep as

```
$ grep 'March\|April\|May'
```

Another example is

```
$ egrep 'M(iss)+ippi'
```

which matches Mississippi as well as Mississississippi.

Another extension in egrep is the + operator. This matches one or more occurrences of the preceding expression. It simplifies those situations where you need to exclude the null string from matching, as is the case with *. For example

```
$ egrep '{[}a-z{]}+'
```

matches one or more lowercase letters and not the null string.

fgrep

The fgrep variant of grep doesn't support regular expressions but does support multiple strings. It's used to search quickly for many different fixed strings. For example, you can put a list of frequently misspelled words into a file and then call fgrep to search for them:

```
$ fgrep -f errors document
```

This will print all lines in *document* that contain one of the strings in the file named *errors*.

Summary

A shell is a programmable and substitutable command line interpreter with many useful and powerful features. The shell is not part of the kernel but instead runs in user space: the layer above it.

Modern shells provide I/O redirection, interactive command line editing, a history mechanism, filename substitution, control flow mechanisms (such as loops, branching, and selection statements, and backgrounding and job control), aliasing, and scripting. Some commands are built into the shell so that files do not need to be loaded to execute them.

All files have three different types of access: read, write, and execute. The permissions for each type of access can be granted independently to the owner of the file, to a group of users associated with the file, and to everyone else. You need to learn a relatively small set of commands in order to perform routine tasks such as navigating through the directory hierarchy, creating and maintaining files and directories, and viewing and changing their permissions.

To learn more about using `bash`, you should read the Bash Reference Manual. You can download the latest version as a PDF from the GNU website (*https://www.gnu.org/home.en.html*). You can also read the `bash` man page or the InfoTex page for `bash`.

Exercises

1. Look at the man page for the `shuf` command, and then write a command that generates a random permutation of the integers from 1 to 100 in a file named *permutation100* in your current working directory.
2. Write a script named *check_procs* that runs the `ps` command every 10 seconds for a total of five times.
3. Modify the preceding script so that it accepts as a command line argument the number of seconds between runs of the `ps` command.
4. Write a sequence of two commands that creates an empty file named *shopping_list* that you can read and write but that cannot be read or written by anyone else.
5. Using nothing but the `cat` command and redirection, show how to add the lines

```
milk
bread
eggs
flour
sugar
```

to the *shopping_list* file. (Hint: Pressing CTRL-D terminates input.)

6. Read the man page for `chmod` and write octal expressions that represent the following permissions.
   (a) Owner can read and write; group can write but not read; others can read.
   (b) Everyone can execute, and the owner can read and write.
   (c) The owner can read; the group can read; others can't do anything.
7. Write a shell script that makes a copy of every file in the current working directory with the same name and an extension *.copy*. For example, the file named *myfile* would have a copy named *myfile.copy*.