2

Shell Programming

Having started this book on programming Linux using C, we now take a detour into writing shell programs. Why? Well, Linux isn't like systems where the command-line interface is an afterthought to the graphical interface. UNIX, Linux's inspiration, originally had no graphical interface at all; everything was done from the command line. Consequently, the command-line system of UNIX underwent a lot of development and became a very powerful feature. This has been carried into Linux, and some of the most powerful things that you can do are most easily done from the shell. Because the shell is so important to Linux, and is so useful for automating simple tasks, shell programming is covered early.

Throughout this chapter, we'll be showing you the syntax, structures, and commands available to you when you're programming the shell, usually making use of interactive (screen-based) examples. These should serve as a useful synopsis of most of the shell's features and their effects. We will also sneak a look at a couple of particularly useful command-line utilities often called from the shell: `grep` and `find`. While looking at `grep`, we also cover the fundamentals of regular expressions, which crop up in Linux utilities and in programming languages such as Perl, Ruby, and PHP. At the end of the chapter, you'll learn how to program a real-life script, which is reprogrammed and extended in C throughout the book. This chapter covers the following:

- What a shell is
- Basic considerations
- The subtleties of syntax: variables, conditions, and program control
- Lists
- Functions
- Commands and command execution
- Here documents
- Debugging
- `grep` and regular expressions
- `find`

Whether you're faced with a complex shell script in your system administration, or you want to prototype your latest big (but beautifully simple) idea, or just want to speed up some repetitive task, this chapter is for you.

Why Program with a Shell?

One reason to use the shell for programming is that you can program the shell quickly and simply. Moreover, a shell is always available even on the most basic Linux installation, so for simple prototyping you can find out if your idea works. The shell is also ideal for any small utilities that perform some relatively simple task for which efficiency is less important than easy configuration, maintenance, and portability. You can use the shell to organize process control, so that commands run in a predetermined sequence dependent on the successful completion of each stage.

Although the shell has superficial similarities to the Windows command prompt, it's much more powerful, capable of running reasonably complex programs in its own right. Not only can you execute commands and call Linux utilities, you can also write them. The shell executes shell programs, often referred to as *scripts*, which are interpreted at runtime. This generally makes debugging easier because you can easily execute single lines, and there's no recompile time. However, this can make the shell unsuitable for time-critical or processor-intensive tasks.

A Bit of Philosophy

Here we come to a bit of UNIX — and of course Linux — philosophy. UNIX is built on and depends on a high level of code reuse. You build a small and simple utility and people use it as one link in a string of others to form a command. One of the pleasures of Linux is the variety of excellent tools available. A simple example is this command:

```
$ ls -al | more
```

This command uses the `ls` and `more` utilities and pipes the output of the file listing to a screen-at-a-time display. Each utility is one more building block. You can often use many small scripts together to create large and complex suites of programs.

For example, if you want to print a reference copy of the `bash manual` pages, then use

```
$ man bash | col -b | lpr
```

Furthermore, because of Linux's automatic file type handling, the users of these utilities usually don't need to know what language the utilities are written in. If the utility needs to run faster, it's quite common to prototype utilities in the shell and reimplement them later in C or C++, Perl, Python, or some other language that executes more swiftly once an idea has proven its worth. Conversely, if the utility works adequately in the shell, you can leave well enough alone.

Whether or not you ever reimplement the script depends on whether it needs optimizing, whether it needs to be portable, whether it should be easy to change, and whether (as usually happens) it outgrows its original purpose.

Numerous examples of shell scripts are already loaded on your Linux system in case you're curious, including package installers, `.xinitrc` *and* `startx`*, and the scripts in* `/etc/rc.d` *to configure the system on boot-up.*

What Is a Shell?

Before jumping in and discussing how to program using a shell, let's review the shell's function and the different shells available for Linux. A *shell* is a program that acts as the interface between you and the Linux system, enabling you to enter commands for the operating system to execute. In that respect, it resembles the Windows command prompt, but as mentioned earlier, Linux shells are much more powerful. For example, input and output can be redirected using < and >, data piped between simultaneously executing programs using `|`, and output from a subprocess grabbed by using `$(...)`. On Linux it's quite feasible to have multiple shells installed, with different users able to pick the one they prefer. Figure 2-1 shows how the shell (two shells actually, both bash and csh) and other programs sit around the Linux kernel.

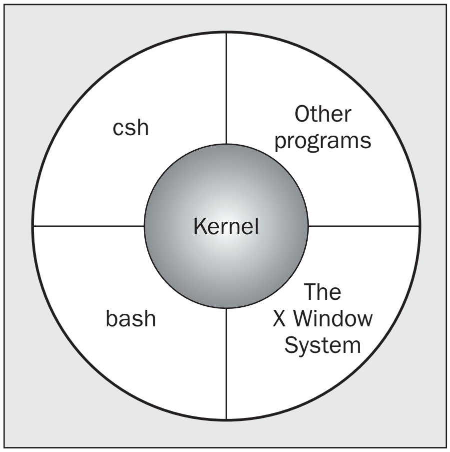

Figure 2-1

Because Linux is so modular, you can slot in one of the many different shells in use, although most of them are derived from the original Bourne shell. On Linux, the standard shell that is always installed as `/bin/sh` is called *bash* (the GNU Bourne-Again SHell), from the GNU suite of tools. Because this is an excellent shell that is always installed on Linux systems, is open source, and is portable to almost all UNIX variants, bash is the shell we will be using. This chapter uses bash version 3 and mostly uses the features common to all POSIX-compatible shells. We assume that the shell has been installed as `/bin/sh` and that it is the default shell for your login. On most Linux distributions, the program `/bin/sh`, the default shell, is actually a link to the program `/bin/bash`.

You can check the version of bash you have with the following command:

```
$ /bin/bash --version
GNU bash, version 3.2.9(1)-release (i686-pc-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
```

To change to a different shell — if bash isn't the default on your system, for example — just execute the desired shell's program (e.g., `/bin/bash`) to run the new shell and change the command prompt. If you are using UNIX, and bash isn't installed, you can download it free from the GNU Web site at `www.gnu.org`. The sources are highly portable, and chances are good that it will compile on your version of UNIX straight out of the box.

When you create Linux users, you can set the shell that they will use, either when the user is created or afterwards by modifying their details. Figure 2-2 shows the selection of the shell for a user using Fedora.

Figure 2-2

Many other shells are available, either free or commercially. The following table offers a brief summary of some of the more common shells available:

Shell Name	A Bit of History
`sh` (Bourne)	The original shell from early versions of UNIX
`csh`, `tcsh`, `zsh`	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
`ksh`, `pdksh`	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
`bash`	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it's not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.

Except for the C shell and a small number of derivatives, all of these are very similar and are closely aligned with the shell specified in the X/Open 4.2 and POSIX 1003.2 specifications. POSIX 1003.2 provides the minimum specification for a shell, but the extended specification in X/Open provides a more friendly and powerful shell. X/Open is usually the more demanding specification, but it also yields a friendlier system.

Pipes and Redirection

Before we get down to the details of shell programs, we need to say a little about how inputs and outputs of Linux programs (not just shell programs) can be redirected.

Redirecting Output

You may already be familiar with some redirection, such as

```
$ ls -l > lsoutput.txt
```

which saves the output of the `ls` command into a file called `lsoutput.txt`.

However, there is much more to redirection than this simple example reveals. You'll learn more about the standard file descriptors in Chapter 3, but for now all you need to know is that file descriptor 0 is the standard input to a program, file descriptor 1 is the standard output, and file descriptor 2 is the standard error output. You can redirect each of these independently. In fact, you can also redirect other file descriptors, but it's unusual to want to redirect any other than the standard ones: 0, 1, and 2.

The preceding example redirects the standard output into a file by using the > operator. By default, if the file already exists, then it will be overwritten. If you want to change the default behavior, you can use the command `set -o noclobber` (or `set -C`), which sets the `noclobber` option to prevent a file from being overwritten using redirection. You can cancel this option using `set +o noclobber`. You'll see more options for the `set` command later in the chapter.

To append to the file, use the >> operator. For example,

```
$ ps >> lsoutput.txt
```

will append the output of the `ps` command to the end of the specified file.

To redirect the standard error output, preface the > operator with the number of the file descriptor you wish to redirect. Because the standard error is on file descriptor 2, use the `2>` operator. This is often useful to discard error information and prevent it from appearing on the screen.

Suppose you want to use the `kill` command to kill a process from a script. There is always a slight risk that the process will die before the `kill` command is executed. If this happens, `kill` will write an error message to the standard error output, which, by default, will appear on the screen. By redirecting both the standard output and the error, you can prevent the `kill` command from writing any text to the screen.

The command

```
$ kill -HUP 1234 >killout.txt 2>killerr.txt
```

will put the output and error information into separate files.

If you prefer to capture both sets of output into a single file, you can use the `>&` operator to combine the two outputs. Therefore,

```
$ kill -1 1234 >killouterr.txt 2>&1
```

will put both the output and error outputs into the same file. Notice the order of the operators. This reads as "redirect standard output to the file `killouterr.txt`, and then direct standard error to the same place as the standard output." If you get the order wrong, the redirect won't work as you expect.

Because you can discover the result of the `kill` command using the return code (discussed in more detail later in this chapter), you don't often want to save either standard output or standard error. You can use the Linux universal "bit bucket" of `/dev/null` to efficiently discard the entire output, like this:

```
$ kill -1 1234 >/dev/null 2>&1
```

Redirecting Input

Rather like redirecting output, you can also redirect input. For example,

```
$ more < killout.txt
```

Obviously, this is a rather trivial example under Linux; the Linux `more` command is quite happy to accept filenames as parameters, unlike the Windows command-line equivalent.

Pipes

You can connect processes using the pipe operator (|). In Linux, unlike in MS-DOS, processes connected by pipes can run simultaneously and are automatically rescheduled as data flows between them. As a simple example, you could use the `sort` command to sort the output from `ps`.

If you don't use pipes, you must use several steps, like this:

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

A much more elegant solution is to connect the processes with a pipe:

```
$ ps | sort > pssort.out
```

Because you probably want to see the output paginated on the screen, you could connect a third process, `more`, all on the same command line:

```
$ ps | sort | more
```

There's practically no limit to the permissible number of connected processes. Suppose you want to see all the different process names that are running excluding shells. You could use

```
$ ps -xo comm | sort | uniq | grep -v sh | more
```

This takes the output of `ps`, sorts it into alphabetical order, extracts processes using `uniq`, uses `grep -v sh` to remove the process named `sh`, and finally displays it paginated on the screen.

As you can see, this is a much more elegant solution than a string of separate commands, each with its own temporary file. However, be wary of one thing here: If you have a string of commands, the output file is created or written to immediately when the set of commands is created, so never use the same filename twice in a string of commands. If you try to do something like

```
cat mydata.txt | sort | uniq > mydata.txt
```

you will end up with an empty file, because you will overwrite the `mydata.txt` file before you read it.

The Shell as a Programming Language

Now that you've seen some basic shell operations, it's time to move on to some actual shell programs. There are two ways of writing shell programs. You can type a sequence of commands and allow the shell to execute them interactively, or you can store those commands in a file that you can then invoke as a program.

Interactive Programs

Just typing the shell script on the command line is a quick and easy way of trying out small code fragments, and is very useful while you are learning or just testing things out.

Suppose you have a large number of C files and wish to examine the files that contain the string `POSIX`. Rather than search using the `grep` command for the string in the files and then list the files individually, you could perform the whole operation in an interactive script like this:

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

Note how the normal `$` shell prompt changes to a `>` when the shell is expecting further input. You can type away, letting the shell decide when you're finished, and the script will execute immediately.

In this example, the `grep` command prints the files it finds containing `POSIX` and then `more` displays the contents of the file to the screen. Finally, the shell prompt returns. Note also that you called the shell variable that deals with each of the files to self-document the script. You could equally well have used `i`, but `file` is more meaningful for humans to read.

The shell also performs wildcard expansion (often referred to as *globbing*). You are almost certainly aware of the use of `'*'` as a wildcard to match a string of characters. What you may not know is that you can request single-character wildcards using `?`, while `[`*`set`*`]` allows any of a number of single characters to be checked. `[^set]` negates the set — that is, it includes anything but the set you've specified. Brace expansion using `{}` (available on some shells, including `bash`) allows you to group arbitrary strings together in a set that the shell will expand. For example,

```
$ ls my_{finger,toe}s
```

will list the files my_fingers and my_toes. This command uses the shell to check every file in the current directory. We will come back to these rules for matching patterns near the end of the chapter when we look in more detail at grep and the power of regular expressions.

Experienced Linux users would probably perform this simple operation in a much more efficient way, perhaps with a command such as

```
$ more `grep -l POSIX *`
```

or the synonymous construction

```
$ more $(grep -l POSIX *)
```

In addition,

```
$ grep -l POSIX * | more
```

will output the name of the file whose contents contained the string POSIX. In this script, you see the shell making use of other commands, such as grep and more, to do the hard work. The shell simply enables you to glue several existing commands together in new and powerful ways. You will see wildcard expansion used many times in the following scripts, and we'll look at the whole area of expansion in more detail when we look at regular expressions in the section on the grep command.

Going through this long rigmarole every time you want to execute a sequence of commands is a bore. You need to store the commands in a file, conventionally referred to as a *shell script*, so you can execute them whenever you like.

Creating a Script

Using any text editor, you need to create a file containing the commands; create a file called first that looks like this:

```
#!/bin/sh

# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    echo $file
  fi
done

exit 0
```

Comments start with a # and continue to the end of a line. Conventionally, though, # is kept in the first column. Having made such a sweeping statement, we next note that the first line, #!/bin/sh, is a special form of comment; the `#!` characters tell the system that the argument that follows on the line is the program to be used to execute this file. In this case, `/bin/sh` is the default shell program.

Note the absolute path specified in the comment. It is conventional to keep this shorter than 32 characters for backward compatibility, because some older UNIX versions can only use this limited number of characters when using `#!`*, although Linux generally does not have this limitation.*

Since the script is essentially treated as standard input to the shell, it can contain any Linux commands referenced by your `PATH` environment variable.

The `exit` command ensures that the script returns a sensible exit code (more on this later in the chapter). This is rarely checked when programs are run interactively, but if you want to invoke this script from another script and check whether it succeeded, returning an appropriate exit code is very important. Even if you never intend to allow your script to be invoked from another, you should still exit with a reasonable code. Have faith in the usefulness of your script: Assume it may need to be reused as part of another script someday.

A zero denotes success in shell programming. Since the script as it stands can't detect any failures, it always returns success. We'll come back to the reasons for using a zero exit code for success later in the chapter, when we look at the `exit` command in more detail.

Notice that this script does not use any filename extension or suffix; Linux, and UNIX in general, rarely makes use of the filename extension to determine the type of a file. You could have used `.sh` or added a different extension, but the shell doesn't care. Most preinstalled scripts will not have any filename extension, and the best way to check if they are scripts or not is to use the `file` command — for example, `file first` or `file /bin/bash`. Use whatever convention is applicable where you work, or suits you.

Making a Script Executable

Now that you have your script file, you can run it in two ways. The simpler way is to invoke the shell with the name of the script file as a parameter:

```
$ /bin/sh first
```

This should work, but it would be much better if you could simply invoke the script by typing its name, giving it the respectability of other Linux commands. Do this by changing the file mode to make the file executable for all users using the `chmod` command:

```
$ chmod +x first
```

Of course, this isn't the only way to use `chmod` to make a file executable. Use `man chmod` to find out more about octal arguments and other options.

You can then execute it using the command

```
$ first
```

You may get an error saying the command wasn't found. This is almost certainly because the shell environment variable `PATH` isn't set to look in the current directory for commands to execute. To change this, either type `PATH=$PATH:.` on the command line or edit your `.bash_profile` file to add this command to the end of the file; then log out and log back in again. Alternatively, type `./first` in the directory containing the script, to give the shell the full relative path to the file.

Specifying the path prepended with `./` does have one other advantage: It ensures that you don't accidentally execute another command on the system with the same name as your script file.

You shouldn't change the `PATH` variable like this for the superuser, conventionally the user name `root`. It's a security loophole, because the system administrator logged in as root can be tricked into invoking a fake version of a standard command. One of the authors admits to doing this once — just to prove a point to the system administrator about security, of course! It's only a slight risk on ordinary accounts to include the current directory in the path, so if you are particularly concerned, just get into the habit of prepending `./` to all commands that are in the local directory.

Once you're confident that your script is executing properly, you can move it to a more appropriate location than the current directory. If the command is just for your own use, you could create a `bin` directory in your home directory and add that to your path. If you want the script to be executable by others, you could use `/usr/local/bin` or another system directory as a convenient location for adding new programs. If you don't have root permissions on your system, you could ask the system administrator to copy your file for you, although you may have to convince them of its worth first. To prevent other users from changing the script, perhaps accidentally, you should remove write access from it. The sequence of commands for the administrator to set ownership and permissions would be something like this:

```
# cp first /usr/local/bin
# chown root /usr/local/bin/first
# chgrp root /usr/local/bin/first
# chmod 755 /usr/local/bin/first
```

Notice that rather than alter a specific part of the permission flags, you use the absolute form of the `chmod` here because you know exactly what permissions you require.

If you prefer, you can use the rather longer, but perhaps more obvious, form of the `chmod` command:

```
# chmod u=rwx,go=rx /usr/local/bin/first
```

Check the manual page of `chmod` for more details.

In Linux you can delete a file if you have write permission on the directory that contains it. To be safe, ensure that only the superuser can write to directories containing files that you want to keep safe. This makes sense because a directory is just another file, and having write permission to a directory file allows users to add and remove names.

Shell Syntax

Now that you've seen an example of a simple shell program, it's time to look in greater depth at the programming power of the shell. The shell is quite an easy programming language to learn, not least because it's easy to test small program fragments interactively before combining them into bigger scripts. You can use the bash shell to write quite large, structured programs. The next few sections cover the following:

- Variables: strings, numbers, environments, and parameters
- Conditions: shell Booleans
- Program control: `if`, `elif`, `for`, `while`, `until`, `case`
- Lists
- Functions
- Commands built into the shell
- Getting the result of a command
- Here documents

Variables

You don't usually declare variables in the shell before using them. Instead, you create them by simply using them (for example, when you assign an initial value to them). By default, all variables are considered and stored as strings, even when they are assigned numeric values. The shell and some utilities will convert numeric strings to their values in order to operate on them as required. Linux is a case-sensitive system, so the shell considers the variable `foo` to be different from `Foo`, and both to be different from `FOO`.

Within the shell you can access the contents of a variable by preceding its name with a `$`. Whenever you extract the contents of a variable, you must give the variable a preceding `$`. When you assign a value to a variable, just use the name of the variable, which is created dynamically if necessary. An easy way to check the contents of a variable is to `echo` it to the terminal, preceding its name with a `$`.

On the command line, you can see this in action when you set and check various values of the variable `salutation`:

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

> **Note how a string must be delimited by quote marks if it contains spaces. In addition, there can't be any spaces on either side of the equals sign.**

You can assign user input to a variable by using the `read` command. This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text. The read normally completes when the user presses Enter. When reading a variable from the terminal, you don't usually need the quote marks:

```
$ read salutation
Wie geht's?
$ echo $salutation
Wie geht's?
```

Quoting

Before moving on, you should be clear about one feature of the shell: the use of quotes.

Normally, parameters in scripts are separated by whitespace characters (e.g., a space, a tab, or a newline character). If you want a parameter to contain one or more whitespace characters, you must quote the parameter.

The behavior of variables such as `$foo` inside quotes depends on the type of quotes you use. If you enclose a `$` variable expression in double quotes, then it's replaced with its value when the line is executed. If you enclose it in single quotes, then no substitution takes place. You can also remove the special meaning of the `$` symbol by prefacing it with a `\`.

Usually, strings are enclosed in double quotes, which protects variables from being separated by white space but allows `$` expansion to take place.

Try It Out Playing with Variables

This example shows the effect of quotes on the output of a variable:

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

This behaves as follows:

```
$ ./variable
Hi there
Hi there
$myvar
$myvar
```

```
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

The variable `myvar` is created and assigned the string `Hi there`. The contents of the variable are displayed with the `echo` command, showing how prefacing the variable with a `$` character expands the contents of the variable. You see that using double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do. You also use the `read` command to get a string from the user.

Environment Variables

When a shell script starts, some variables are initialized from values in the environment. These are normally in all uppercase form to distinguish them from user-defined (shell) variables in scripts, which are conventionally lowercase. The variables created depend on your personal configuration. Many are listed in the manual pages, but the principal ones are listed in the following table:

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$PS1`	A command prompt, frequently `$`, but in bash you can use some more complex values; for example, the string `[\u@\h \W]$` is a popular default that tells you the user, machine name, and current directory, as well as providing a $ prompt.
`$PS2`	A secondary prompt, used when prompting for additional input; usually `>`.
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

If you want to check out how the program works in a different environment by running the `env <command>`, try looking at the `env` manual pages. Later in the chapter you'll see how to set environment variables in subshells using the `export` command.

Parameter Variables

If your script is invoked with parameters, some additional variables are created. If no parameters are passed, the environment variable `$#` still exists but has a value of 0.

The parameter variables are listed in the following table:

Parameter Variable	Description
`$1, $2, …`	The parameters given to the script
`$*`	A list of all the parameters, in a single variable, separated by the first character in the environment variable `IFS`. If `IFS` is modified, then the way `$*` separates the command line into parameters will change.
`$@`	A subtle variation on `$*`; it doesn't use the `IFS` environment variable, so parameters are not run together even if `IFS` is empty.

It's easy to see the difference between `$@` and `$*` by trying them out:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

As you can see, within double quotes, `$@` expands the positional parameters as separate fields, regardless of the `IFS` value. In general, if you want access to the parameters, `$@` is the sensible choice.

In addition to printing the contents of variables using the `echo` command, you can also read them by using the `read` command.

Try It Out Manipulating Parameter and Environment Variables

The following script demonstrates some simple variable manipulation. Once you've typed the script and saved it as `try_var`, don't forget to make it executable with `chmod +x try_var`.

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"
```

```
echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

If you run this script, you get the following output:

```
$ ./try_var foo bar baz
Hello
The program ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

How It Works

This script creates the variable `salutation`, displays its contents, and then shows how various parameter variables and the environment variable `$HOME` already exist and have appropriate values.

We'll return to parameter substitution in more detail later in the chapter.

Conditions

Fundamental to all programming languages is the ability to test conditions and perform different actions based on those decisions. Before we talk about that, though, let's look at the conditional constructs that you can use in shell scripts and then examine the control structures that use them.

A shell script can test the exit code of any command that can be invoked from the command line, including the scripts that you have written yourself. That's why it's important to always include an `exit` command with a value at the end of any scripts that you write.

The test or [Command

In practice, most scripts make extensive use of the `[` or `test` command, the shell's Boolean check. On some systems, the `[` and `test` commands are synonymous, except that when the `[` command is used, a trailing `]` is also used for readability. Having a `[` command might seem a little odd, but within the code it does make the syntax of commands look simple, neat, and more like other programming languages.

These commands call an external program in some older UNIX shells, but they tend to be built in to more modern ones. We'll come back to this when we look at commands in a later section.

Because the `test` command is infrequently used outside shell scripts, many Linux users who have never written shell scripts try to write simple programs and call them *test*. If such a program doesn't work, it's probably conflicting with the shell's `test` command. To find out whether your system has an external command of a given name, try typing something like `which test`, to check which `test` command is being executed, or use `./test` to ensure that you execute the script in the current directory. When in doubt, just get into the habit of executing your scripts by preceding the script name with `./` when invoking them.

We'll introduce the `test` command using one of the simplest conditions: checking to see whether a file exists. The command for this is `test -f <filename>`, so within a script you can write

```
if test -f fred.c
then
...
fi
```

You can also write it like this:

```
if [ -f fred.c ]
then
...
fi
```

The `test` command's exit code (whether the condition is satisfied) determines whether the conditional code is run.

Note that you must put spaces between the `[` braces and the condition being checked. You can remember this by remembering that `[` is just the same as writing `test`, and you would always leave a space after the `test` command.

If you prefer putting `then` on the same line as `if`, you must add a semicolon to separate the test from the `then`:

```
if [ -f fred.c ]; then
...
fi
```

The condition types that you can use with the `test` command fall into three types: *string comparison, arithmetic comparison,* and *file conditionals*. The following table describes these condition types:

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)
Arithmetic Comparison	**Result**
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`
`! expression`	True if the expression is false, and vice versa
File Conditional	**Result**
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

> **You may be wondering what the `set-group-id` and `set-user-id` (also known as `set-gid` and `set-uid`) bits are. The `set-uid` bit gives a program the permissions of its owner, rather than its user, while the `set-gid` bit gives a program the permissions of its group. The bits are set with `chmod`, using the `s` and `g` options. The `set-gid` and `set-uid` flags have no effect on files containing shell scripts, only on executable binary files.**

We're getting ahead of ourselves slightly, but following is an example of how you would test the state of the file `/bin/bash`, just so you can see what these look like in use:

```
#!/bin/sh

if [ -f /bin/bash ]
then
  echo "file /bin/bash exists"
fi

if [ -d /bin/bash ]
then
  echo "/bin/bash is a directory"
else
  echo "/bin/bash is NOT a directory"
fi
```

Before the test can be true, all the file conditional tests require that the file also exists. This list contains just the more commonly used options to the `test` command, so for a complete list refer to the manual entry. If you're using bash, where `test` is built in, use the `help test` command to get more details. We'll use some of these options later in the chapter.

Now that you know about conditions, you can look at the control structures that use them.

Control Structures

The shell has a set of control structures, which are very similar to other programming languages.

> **In the following sections, the statements are the series of commands to perform when, while, or until the condition is fulfilled.**

if

The `if` statement is very simple: It tests the result of a command and then conditionally executes a group of statements:

```
if condition
then
  statements
```

```
else
  statements
fi
```

A common use for `if` is to ask a question and then make a decision based on the answer:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

This would give the following output:

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

This script uses the `[` command to test the contents of the variable `timeofday`. The result is evaluated by the `if` command, which then allows different lines of code to be executed.

> **Notice that you use extra white space to indent the statements inside the `if`. This is just a convenience for the human reader; the shell ignores the additional white space.**

elif

Unfortunately, there are several problems with this very simple script. For one thing, it will take any answer except `yes` as meaning `no`. You can prevent this by using the `elif` construct, which allows you to add a second condition to be checked when the `else` portion of the `if` is executed.

Try It Out Doing Checks with an elif

You can modify the previous script so that it reports an error message if the user types in anything other than `yes` or `no`. Do this by replacing the `else` with `elif` and then adding another condition:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
  echo "Good morning"
```

```
elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

How It Works

This is quite similar to the previous example, but now the `elif` command tests the variable again if the first `if` condition is not true. If neither of the tests is successful, an error message is printed and the script exits with the value 1, which the caller can use in a calling program to check whether the script was successful.

A Problem with Variables

This fixes the most obvious defect, but a more subtle problem is lurking. Try this new script, but just press Enter (or Return on some keyboards), rather than answering the question. You'll get this error message:

```
[: =: unary operator expected
```

What went wrong? The problem is in the first `if` clause. When the variable `timeofday` was tested, it consisted of a blank string. Therefore, the `if` clause looks like

```
if [ = "yes" ]
```

which isn't a valid condition. To avoid this, you must use quotes around the variable:

```
if [ "$timeofday" = "yes" ]
```

An empty variable then gives the valid test:

```
if [ "" = "yes" ]
```

The new script is as follows:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]
then
  echo "Good morning"
elif [ "$timeofday" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
```

```
  exit 1
fi

exit 0
```

This is safe should a user just press Enter in answer to the question.

> **If you want the `echo` command to delete the trailing new line, the most portable option is to use the `printf` command (see the `printf` section later in this chapter), rather than the `echo` command. Some shells use `echo -e`, but that's not supported on all systems. bash allows `echo -n` to suppress the new line, so if you are confident your script needs to work only on bash, we suggest using that syntax.**

```
echo -n "Is it morning? Please answer yes or no: "
```

Note that you need to leave an extra space before the closing quotes so that there is a gap before the user-typed response, which looks neater.

for

Use the `for` construct to loop through a range of values, which can be any set of strings. They could be simply listed in the program or, more commonly, the result of a shell expansion of filenames.

The syntax is simple:

```
for variable in values
do
  statements
done
```

Try It Out Using a for Loop with Fixed Strings

The values are normally strings, so you can write the following:

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done
exit 0
```

That results in the following output:

```
bar
fud
43
```

> **What would happen if you changed the first line from** `for foo in bar fud 43` **to** `for foo in "bar fud 43"`**? Remember that adding the quotes tells the shell to consider everything between them as a single string. This is one way of getting spaces to be stored in a variable.**

How It Works

This example creates the variable `foo` and assigns it a different value each time around the `for` loop. Since the shell considers all variables to contain strings by default, it's just as valid to use the string `43` as the string `fud`.

Try It Out — Using a for Loop with Wildcard Expansion

As mentioned earlier, it's common to use the `for` loop with a shell expansion for filenames. This means using a wildcard for the string value and letting the shell fill out all the values at run time.

You've already seen this in the original example, `first`. The script used shell expansion, the `*` expanding to the names of all the files in the current directory. Each of these in turn is used as the variable `$file` inside the `for` loop.

Let's quickly look at another wildcard expansion. Imagine that you want to print all the script files starting with the letter "f" in the current directory, and you know that all your scripts end in `.sh`. You could do it like this:

```
#!/bin/sh

for file in $(ls f*.sh); do
  lpr $file
done
exit 0
```

How It Works

This illustrates the use of the `$(command)` syntax, which is covered in more detail later (in the section on command execution). Basically, the parameter list for the `for` command is provided by the output of the command enclosed in the `$()` sequence.

The shell expands `f*.sh` to give the names of all the files matching this pattern.

> **Remember that all expansion of variables in shell scripts is done when the script is executed, never when it's written, so syntax errors in variable declarations are found only at execution time, as shown earlier when we were quoting empty variables.**

while

Because all shell values are considered strings by default, the `for` loop is good for looping through a series of strings, but is not so useful when you don't know in advance how many times you want the loop to be executed.

When you need to repeat a sequence of commands, but don't know in advance how many times they should execute, you will normally use a `while` loop, which has the following syntax:

```
while condition do
  statements
done
```

For example, here is a rather poor password-checking program:

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done
exit 0
```

An example of the output from this script is as follows:

```
Enter password
password
Sorry, try again
secret
$
```

Clearly, this isn't a very secure way of asking for a password, but it does serve to illustrate the `while` statement. The statements between `do` and `done` are continuously executed until the condition is no longer true. In this case, you're checking whether the value of `trythis` is equal to `secret`. The loop will continue until `$trythis` equals `secret`. You then continue executing the script at the statement immediately following the `done`.

until

The `until` statement has the following syntax:

```
until condition
do
  statements
done
```

This is very similar to the `while` loop, but with the condition test reversed. In other words, the loop continues until the condition becomes true, not while the condition is true.

> **In general, if a loop should always execute at least once, use a `while` loop; if it may not need to execute at all, use an `until` loop.**

As an example of an `until` loop, you can set up an alarm that is initiated when another user, whose login name you pass on the command line, logs on:

```
#!/bin/bash

until who | grep "$1" > /dev/null
do
   sleep 60
done

# now ring the bell and announce the expected user.

echo -e '\a'
echo "**** $1 has just logged in ****"

exit 0
```

If the user is already logged on, the loop doesn't need to execute at all, so using `until` is a more natural choice than `while`.

case

The `case` construct is a little more complex than those you have encountered so far. Its syntax is as follows:

```
case variable in
  pattern [ | pattern] ...) statements;;
  pattern [ | pattern] ...) statements;;
  ...
esac
```

This may look a little intimidating, but the `case` construct enables you to match the contents of a variable against patterns in quite a sophisticated way and then allows execution of different statements, depending on which pattern was matched. It is much simpler than the alternative way of checking several conditions, which would be to use multiple `if`, `elif`, and `else` statements.

Notice that each pattern line is terminated with double semicolons (`;;`). You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.

The capability to match multiple patterns and then execute multiple related statements makes the `case` construct a good way of dealing with user input. The best way to see how `case` works is with an example. We'll develop it over three Try It Out examples, improving the pattern matching each time.

> **Be careful with the `case` construct if you are using wildcards such as '*' in the pattern. The problem is that the first matching pattern will be taken, even if a later pattern matches more exactly.**

Try It Out Case I: User Input

You can write a new version of the input-testing script and, using the `case` construct, make it a little more selective and forgiving of unexpected input:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes)  echo "Good Morning";;
    no )  echo "Good Afternoon";;
    y  )  echo "Good Morning";;
    n  )  echo "Good Afternoon";;
    *  )  echo "Sorry, answer not recognized";;
esac

exit 0
```

How It Works

When the `case` statement is executing, it takes the contents of `timeofday` and compares it to each string in turn. As soon as a string matches the input, the `case` command executes the code following the `)` and finishes.

The `case` command performs normal expansion on the strings that it's using for comparison. You can therefore specify part of a string followed by the * wildcard. Using a single `*` will match all possible strings, so always put one after the other matching strings to ensure that the `case` statement ends with some default action if no other strings are matched. This is possible because the `case` statement compares against each string in turn. It doesn't look for a best match, just the first match. The default condition often turns out to be the impossible condition, so using `*` can help in debugging scripts.

Try It Out Case II: Putting Patterns Together

The preceding `case` construction is clearly more elegant than the multiple `if` statement version, but by putting the patterns together, you can make a much cleaner version:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )      echo "Good Morning";;
    n* | N* )                  echo "Good Afternoon";;
    * )                        echo "Sorry, answer not recognized";;
esac

exit 0
```

How It Works

This script uses multiple strings in each entry of the `case` so that `case` tests several different strings for each possible statement. This makes the script both shorter and, with practice, easier to read. This code also shows how the `*` wildcard can be used, although this may match unintended patterns. For example, if the user enters `never`, then this will be matched by `n*` and `Good Afternoon` will be displayed, which isn't the intended behavior. Note also that the `*` wildcard expression doesn't work within quotes.

Try It Out Case III: Executing Multiple Statements

Finally, to make the script reusable, you need to have a different exit value when the default pattern is used because the input was not understood:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )
           echo "Good Morning"
           echo "Up bright and early this morning"
           ;;
    [nN]*)
           echo "Good Afternoon"
           ;;
    *)
           echo "Sorry, answer not recognized"
           echo "Please answer yes or no"
           exit 1
           ;;
esac

exit 0
```

How It Works

To show a different way of pattern matching, this code changes the way in which the `no` case is matched. You also see how multiple statements can be executed for each pattern in the `case` statement. You must be careful to put the most explicit matches first and the most general match last. This is important because `case` executes the first match it finds, not the best match. If you put the `*)` first, it would always be matched, regardless of what was input.

Note that the `;;` before `esac` is optional. Unlike C programming, where leaving out a break is poor programming practice, leaving out the final `;;` is no problem if the last case is the default because no other cases will be considered.

To make the `case` matching more powerful, you could use something like this:

```
[yY] | [Yy][Ee][Ss] )
```

This restricts the permitted letters while allowing a variety of answers, and offers more control than the `*` wildcard.

Lists

Sometimes you want to connect commands in a series. For instance, you may want several different conditions to be met before you execute a statement:

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
        fi
    fi
fi
```

Or you might want at least one of a series of conditions to be true:

```
if [ -f this_file ]; then
   foo="True"
elif [ -f that_file ]; then
   foo="True"
elif [ -f the_other_file ]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo" = "True" ]; then
   echo "One of the files exists"
fi
```

Although these can be implemented using multiple `if` statements, you can see that the results are awkward. The shell has a special pair of constructs for dealing with lists of commands: the AND list and the OR list. These are often used together, but we'll review their syntax separately.

The AND List

The AND list construct enables you to execute a series of commands, executing the next command only if all the previous commands have succeeded. The syntax is

```
statement1 && statement2 && statement3 && ...
```

Starting at the left, each statement is executed; if it returns `true`, the next statement to the right is executed. This continues until a statement returns `false`, after which no more statements in the list are executed. The `&&` tests the condition of the preceding command.

Each statement is executed independently, enabling you to mix many different commands in a single list, as the following script shows. The AND list as a whole succeeds if all commands are executed successfully, but it fails otherwise.

Try It Out AND Lists

In the following script, you touch file_one (to check whether it exists and create it if it doesn't) and then remove file_two. Then the AND list tests for the existence of each of the files and echoes some text in between.

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

Try the script and you'll get the following result:

```
hello
in else
```

How It Works

The touch and rm commands ensure that the files in the current directory are in a known state. The && list then executes the [-f file_one] statement, which succeeds because you just made sure that the file existed. Because the previous statement succeeded, the echo command is executed. This also succeeds (echo always returns true). The third test, [-f file_two], is then executed. It fails because the file doesn't exist. Because the last command failed, the final echo statement isn't executed. The result of the && list is false because one of the commands in the list failed, so the if statement executes its else condition.

The OR List

The OR list construct enables us to execute a series of commands until one succeeds, and then not execute any more. The syntax is as follows:

```
statement1 || statement2 || statement3 || ...
```

Starting at the left, each statement is executed. If it returns false, then the next statement to the right is executed. This continues until a statement returns true, at which point no more statements are executed.

The || list is very similar to the && list, except that the rule for executing the next statement is that the previous statement must fail.

Try It Out — OR Lists

Copy the previous example and change the shaded lines in the following listing:

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

This results in the following output:

```
hello
in if
```

How It Works

The first two lines simply set up the files for the rest of the script. The first command, `[ -f file_one ]`, fails because the file doesn't exist. The `echo` statement is then executed. Surprise, surprise — this returns `true`, and no more commands in the `||` list are executed. The `if` succeeds because one of the commands in the `||` list (the `echo`) was `true`.

The result of both of these constructs is the result of the last statement to be executed.

These list-type constructs execute in a similar way to those in C when multiple conditions are being tested. Only the minimum number of statements is executed to determine the result. Statements that can't affect the result are not executed. This is commonly referred to as *short circuit evaluation*.

Combining these two constructs is a logician's heaven. Try out the following:

```
[ -f file_one ] && command for true || command for false
```

This will execute the first command if the test succeeds and the second command otherwise. It's always best to experiment with these more unusual lists, and in general you should use braces to force the order of evaluation.

Statement Blocks

If you want to use multiple statements in a place where only one is allowed, such as in an AND or OR list, you can do so by enclosing them in braces `{}` to make a statement block. For example, in the application presented later in this chapter, you'll see the following code:

```
get_confirm && {
   grep -v "$cdcatnum" $tracks_file > $temp_file
```

```
    cat $temp_file > $tracks_file
    echo
    add_record_tracks
}
```

Functions

You can define functions in the shell; and if you write shell scripts of any size, you'll want to use them to structure your code.

> **As an alternative, you could break a large script into lots of smaller scripts, each of which performs a small task. This has some drawbacks: Executing a second script from within a script is much slower than executing a function. It's more difficult to pass back results, and there can be a very large number of small scripts. You should consider the smallest part of your script that sensibly stands alone and use that as your measure of when to break a large script into a collection of smaller ones.**

To define a shell function, simply write its name followed by empty parentheses and enclose the statements in braces:

```
function_name () {
  statements
}
```

Try It Out   A Simple Function

Let's start with a really simple function:

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

Running the script will output the following:

```
script starting
Function foo is executing
script ending
```

How It Works

This script starts executing at the top, so nothing is different there, but when it finds the `foo() {` construct, it knows that a function called `foo` is being defined. It stores the fact that `foo` refers to a function and continues executing after the matching `}`. When the single line `foo` is executed, the shell knows to execute the previously defined function. When this function completes, execution resumes at the line after the call to `foo`.

You must always define a function before you can invoke it, a little like the Pascal style of function definition before invocation, except that there are no forward declarations in the shell. This isn't a problem, because all scripts start executing at the top, so simply putting all the functions before the first call of any function will always cause all functions to be defined before they can be invoked.

When a function is invoked, the positional parameters to the script, `$*`, `$@`, `$#`, `$1`, `$2`, and so on, are replaced by the parameters to the function. That's how you read the parameters passed to the function. When the function finishes, they are restored to their previous values.

> **Some older shells may not restore the value of positional parameters after functions execute. It's wise not to rely on this behavior if you want your scripts to be portable.**

You can make functions return numeric values using the `return` command. The usual way to make functions return strings is for the function to store the string in a variable, which can then be used after the function finishes. Alternatively, you can `echo` a string and catch the result, like this:

```
foo () { echo JAY;}

...

result="$(foo)"
```

Note that you can declare local variables within shell functions by using the `local` keyword. The variable is then only in scope within the function. Otherwise, the function can access the other shell variables that are essentially global in scope. If a local variable has the same name as a global variable, it overlays that variable, but only within the function. For example, you can make the following changes to the preceding script to see this in action:

```
#!/bin/sh

sample_text="global variable"

foo() {

    local sample_text="local variable"
    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text
```

```
foo

echo "script ended"
echo $sample_text

exit 0
```

In the absence of a `return` command specifying a return value, a function returns the exit status of the last command executed.

Try It Out — Returning a Value

The next script, `my_name`, shows how parameters to a function are passed and how functions can return a `true` or `false` result. You call this script with a parameter of the name you want to use in the question.

1. After the shell header, define the function `yes_or_no`:

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}
```

2. Then the main part of the program begins:

```
echo "Original parameters are $*"

if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
```

Typical output from this script might be as follows:

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
```

```
Enter yes or no: yes
Hi Rick, nice name
$
```

How It Works

As the script executes, the function `yes_or_no` is defined but not yet executed. In the `if` statement, the script executes the function `yes_or_no`, passing the rest of the line as parameters to the function after substituting the `$1` with the first parameter to the original script, `Rick`. The function uses these parameters, which are now stored in the positional parameters `$1`, `$2`, and so on, and returns a value to the caller. Depending on the return value, the `if` construct executes the appropriate statement.

As you've seen, the shell has a rich set of control structures and conditional statements. You need to learn some of the commands that are built into the shell; then you'll be ready to tackle a real programming problem with no compiler in sight!

Commands

You can execute two types of commands from inside a shell script. There are "normal" commands that you could also execute from the command prompt (called *external commands*), and there are "built-in" commands (called *internal commands*), as mentioned earlier. Built-in commands are implemented internally to the shell and can't be invoked as external programs. However, most internal commands are also provided as standalone programs — this requirement is part of the POSIX specification. It generally doesn't matter if the command is internal or external, except that internal commands execute more efficiently.

Here we'll cover only the main commands, both internal and external, that we use when we're programming scripts. As a Linux user, you probably know many other commands that are valid at the command prompt. Always remember that you can use any of these in a script in addition to the built-in commands presented here.

break

Use `break` for escaping from an enclosing `for`, `while`, or `until` loop before the controlling condition has been met. You can give `break` an additional numeric parameter, which is the number of loops to break out of, but this can make scripts very hard to read, so we don't suggest you use it. By default, `break` escapes a single level.

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4

for file in fred*
do
    if [ -d "$file" ]; then
        break;
```

```
    fi
done

echo first directory starting fred was $file

rm -rf fred*
exit 0
```

The : Command

The colon command is a null command. It's occasionally useful to simplify the logic of conditions, being an alias for `true`. Since it's built-in, `:` runs faster than `true`, though its output is also much less readable.

You may see it used as a condition for `while` loops; `while :` implements an infinite loop in place of the more common `while true`.

The `:` construct is also useful in the conditional setting of variables. For example,

```
: ${var:=value}
```

Without the `:`, the shell would try to evaluate `$var` as a command.

> **In some, mostly older, shell scripts, you may see the colon used at the start of a line to introduce a comment, but modern scripts should always use # to start a comment line because this executes more efficiently.**

```
#!/bin/sh

rm -f fred
if [ -f fred ]; then
    :
else
    echo file fred did not exist
fi

exit 0
```

continue

Rather like the C statement of the same name, this command makes the enclosing `for`, `while`, or `until` loop continue at the next iteration, with the loop variable taking the next value in the list:

```
#!/bin/sh

rm -rf fred*
echo > fred1
echo > fred2
mkdir fred3
echo > fred4
```

```
for file in fred*
do
    if [ -d "$file" ]; then
          echo "skipping directory $file"
        continue
    fi
    echo file is $file
done

rm -rf fred*
exit 0
```

`continue` can take the enclosing loop number at which to resume as an optional parameter so that you can partially jump out of nested loops. This parameter is rarely used, as it often makes scripts much harder to understand. For example,

```
for x in 1 2 3
do
  echo before $x
  continue 1
  echo after $x
done
```

The output for the preceding will be

```
before 1
before 2
before 3
```

The . Command

The dot (`.`) command executes the command in the current shell:

```
. ./shell_script
```

Normally, when a script executes an external command or script, a new environment (a subshell) is created, the command is executed in the new environment, and the environment is then discarded apart from the exit code that is returned to the parent shell. However, the external `source` and the dot command (two more synonyms) run the commands listed in a script in the same shell that called the script.

Because, by default, a new environment is created when a shell script is executed, any changes to environment variables that the script makes are lost. The dot command, on the other hand, allows the executed script to change the current environment. This is often useful when you use a script as a wrapper to set up your environment for the later execution of some other command. For example, when you're working on several different projects at the same time, you may find you need to invoke commands with different parameters, perhaps to invoke an older version of the compiler for maintaining an old program.

In shell scripts, the dot command works a little like the `#include` directive in C or C++. Though it doesn't literally include the script, it does execute the command in the current context, so you can use it to incorporate variable and function definitions into a script.

Try It Out The Dot Command

The following example uses the dot command on the command line, but you can just as well use it within a script:

1. Suppose you have two files containing the environment settings for two different development environments. To set the environment for the old, classic commands, `classic_set`, you could use the following:

```
#!/bin/sh

version=classic
PATH=/usr/local/old_bin:/usr/bin:/bin:.
PS1="classic> "
```

2. For the new commands, use `latest_set`:

```
#!/bin/sh

version=latest
PATH=/usr/local/new_bin:/usr/bin:/bin:.
PS1=" latest version> "
```

You can set the environment by using these scripts in conjunction with the dot command, as in the following sample session:

```
$ . ./classic_set
classic> echo $version
classic
classic> . /latest_set
latest version> echo $version
latest
latest version>
```

How It Works

The scripts are executed using the dot command, so each script is executed in the current shell. This enables the script to change environment settings in the current shell, which remains changed even when the script finishes executing.

---

echo

Despite the X/Open exhortation to use the `printf` command in modern shells, we've been following common practice by using the `echo` command to output a string followed by a newline character.

A common problem is how to suppress the newline character. Unfortunately, different versions of UNIX have implemented different solutions. The common method in Linux is to use

```
echo -n "string to output"
```

but you'll often come across

```
echo -e "string to output\c"
```

The second option, `echo -e`, ensures that the interpretation of backslashed escape characters, such as `\c` for suppressing a newline, `\t` for outputting a tab and `\n` for outputting carriage returns, is enabled. In older versions of bash this was often set by default, but more recent versions often default to not interpreting backslashed escape characters. See the manual pages for details of the behavior on your distribution.

> **If you need a portable way to remove the trailing newline, you can use the external `tr` command to get rid of it, but it will execute somewhat more slowly. If you need portability to UNIX systems, it's generally better to stick to `printf` if you need to lose the newline. If your scripts need to work only on Linux and bash, `echo -n` should be fine, though you may need to start the file with `#!/bin/bash`, to make it explicit that you desire bash-style behavior.**

eval

The `eval` command enables you to evaluate arguments. It's built into the shell and doesn't normally exist as a separate command. It's probably best demonstrated with a short example borrowed from the X/Open specification itself:

```
foo=10
x=foo
y='$'$x
echo $y
```

This gives the output `$foo`. However,

```
foo=10
x=foo
eval y='$'$x
echo $y
```

gives the output `10`. Thus, `eval` is a bit like an extra `$`: It gives you the value of the value of a variable.

The `eval` command is very useful, enabling code to be generated and run on-the-fly. It does complicate script debugging, but it enables you to do things that are otherwise difficult or even impossible.

exec

The `exec` command has two different uses. Its typical use is to replace the current shell with a different program. For example,

```
exec wall "Thanks for all the fish"
```

in a script will replace the current shell with the `wall` command. No lines in the script after the `exec` will be processed, because the shell that was executing the script no longer exists.

The second use of `exec` is to modify the current file descriptors:

```
exec 3< afile
```

This causes file descriptor three to be opened for reading from file `afile`. It's rarely used.

exit n

The `exit` command causes the script to exit with exit code `n`. If you use it at the command prompt of any interactive shell, it will log you out. If you allow your script to exit without specifying an exit status, then the status of the last command executed in the script is used as the return value. It's always good practice to supply an exit code.

In shell script programming, exit code 0 is success, and codes 1 through 125, inclusive, are error codes that can be used by scripts. The remaining values have reserved meanings, as shown in the following table:

Exit Code	Description
126	The file was not executable
127	A command was not found
128 and above	A signal occurred

Using zero as success may seem a little unusual to many C or C++ programmers. The big advantage in scripts is that they enable you to use 125 user-defined error codes without the need for a global error code variable.

Here's a simple example that returns success if a file called `.profile` exists in the current directory:

```
#!/bin/sh

if [ -f .profile ]; then
    exit 0
fi

exit 1
```

If you're a glutton for punishment, or at least for terse scripts, you can rewrite this script using the combined AND and OR list shown earlier, all on one line:

```
[ -f .profile ] && exit 0 || exit 1
```

export

The `export` command makes the variable named as its parameter available in subshells. By default, variables created in a shell are not available in further (sub)shells invoked from that shell. The `export` command creates an environment variable from its parameter that can be seen by other scripts and programs invoked from the current program. More technically, the exported variables form the environment variables in any child processes derived from the shell. This is best illustrated with an example of two scripts, `export1` and `export2`.

Try It Out Exporting Variables

1. First, list `export2`:

```
#!/bin/sh

echo "$foo"
echo "$bar"
```

2. Now for `export1`. At the end of this script, invoke `export2`:

```
#!/bin/sh

foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"

export2
```

If you run these, you get the following:

```
$ ./export1

The second meta-syntactic variable
$
```

How It Works

The `export2` script simply echoes the values of the two variables. The `export1` script sets both the variables, but only marks `bar` as exported, so when it subsequently invokes `export1`, the value of `foo` has been lost, but the value of `bar` has been exported to the second script. The blank line occurs because `$foo` evaluated to nothing, and echoing a `null` variable gives a newline.

Once a variable has been exported from a shell, it's exported to any scripts invoked from that shell and to any shell they invoke in turn, and so on. If the script `export2` called another script, it would also have the value of `bar` available to it.

> **The commands `set -a` or `set -allexport` will export all variables thereafter.**

---

expr

The `expr` command evaluates its arguments as an expression. It's most commonly used for simple arithmetic in the following form:

```
x=`expr $x + 1`
```

The `` (backtick) characters make x take the result of executing the command expr $x + 1. You could also write it using the syntax $( ) rather than backticks, like this:

```
x=$(expr $x + 1)
```

The expr command is powerful and can perform many expression evaluations. The principal ones are shown in the following table:

Expression Evaluation	Description
expr1 | expr2	expr1 if expr1 is nonzero, otherwise expr2
expr1 & expr2	Zero if either expression is zero, otherwise expr1
expr1 = expr2	Equal
expr1 > expr2	Greater than
expr1 >= expr2	Greater than or equal to
expr1 < expr2	Less than
expr1 <= expr2	Less than or equal to
expr1 != expr2	Not equal
expr1 + expr2	Addition
expr1 - expr2	Subtraction
expr1 * expr2	Multiplication
expr1 / expr2	Integer division
expr1 % expr2	Integer modulo

In newer scripts, the use of expr is normally replaced with the more efficient $((…)) syntax, which is covered later in the chapter.

printf

The printf command is available only in more recent shells. X/Open suggests that it should be used in preference to echo for generating formatted output, though few people seem to follow this advice.

The syntax is

```
printf "format string" parameter1 parameter2 ...
```

The format string is very similar to that used in C or C++, with some restrictions. Principally, floating point isn't supported, because all arithmetic in the shell is performed as integers. The format string consists of any combination of literal characters, escape sequences, and conversion specifiers. All characters in the format string other than % and \ appear literally in the output.

The following escape sequences are supported:

Escape Sequence	Description
`\"`	Double quote
`\\`	Backslash character
`\a`	Alert (ring the bell or beep)
`\b`	Backspace character
`\c`	Suppress further output
`\f`	Form feed character
`\n`	Newline character
`\r`	Carriage return
`\t`	Tab character
`\v`	Vertical tab character
`\ooo`	The single character with octal value `ooo`
`\xHH`	The single character with the hexadecimal value `HH`

The conversion specifier is quite complex, so we list only the common usage here. More details can be found in the bash online manual or in the `printf` pages from section 1 of the online manual (`man 1 printf`). (If you can't find it in section 1 of the manual, try section 3 as an alternative). The conversion specifier consists of a `%` character, followed by a conversion character. The principal conversions are shown in the following table:

Conversion Specifier	Description
`D`	Output a decimal number.
`C`	Output a character.
`S`	Output a string.
`%`	Output the `%` character.

The format string is then used to interpret the remaining parameters and output the result, as shown in the following example,

```
$ printf "%s\n" hello
hello
$ printf "%s %d\t%s" "Hi There" 15 people
Hi There 15   people
```

Notice you must use `" "` to protect the `Hi There` string and make it a single parameter.

return

The `return` command causes functions to return, as mentioned when we looked at functions earlier. `return` takes a single numeric parameter that is available to the script calling the function. If no parameter is specified, then `return` defaults to the exit code of the last command.

set

The `set` command sets the parameter variables for the shell. It can be a useful way of using fields in commands that output space-separated values.

Suppose you want to use the name of the current month in a shell script. The system provides a `date` command, which contains the month as a string, but you need to separate it from the other fields. You can do this using a combination of the `set` command and the `$(...)` construct to execute the `date` command and return the result (described in more detail very soon). The `date` command output has the month string as its second parameter:

```
#!/bin/sh

echo the date is $(date)
set $(date)
echo The month is $2

exit 0
```

This program sets the parameter list to the `date` command's output and then uses the positional parameter `$2` to get at the month.

Notice that we used the `date` command as a simple example to show how to extract positional parameters. Since the `date` command is sensitive to the language locale, in reality you would have extracted the name of the month using `date +%B`. The `date` command has many other formatting options; see the manual page for more details.

You can also use the `set` command to control the way the shell executes by passing it parameters. The most commonly used form of the command is `set -x`, which makes a script display a trace of its currently executing command. We discuss `set` and more of its options when we look at debugging, later in the chapter.

shift

The `shift` command moves all the parameter variables down by one, so that `$2` becomes `$1`, `$3` becomes `$2`, and so on. The previous value of `$1` is discarded, while `$0` remains unchanged. If a numerical parameter is specified in the call to `shift`, the parameters move that many spaces. The other variables, `$*`, `$@`, and `$#`, are also modified in line with the new arrangement of parameter variables.

`shift` is often useful for scanning through parameters passed into a script, and if your script requires 10 or more parameters, you'll need `shift` to access the tenth and beyond.

For example, you can scan through all the positional parameters like this:

```
#!/bin/sh

while [ "$1" != "" ]; do
    echo "$1"
    shift
done

exit 0
```

trap

The `trap` command is used to specify the actions to take on receipt of signals, which you'll meet in more detail later in the book. A common use is to tidy up a script when it is interrupted. Historically, shells always used numbers for the signals, but new scripts should use names taken from the `#include` file `signal.h`, with the `SIG` prefix omitted. To see the signal numbers and associated names, you can just type `trap -l` at a command prompt.

> **For those not familiar with signals, they are events sent asynchronously to a program. By default, they normally cause the program to terminate.**

The `trap` command is passed the action to take, followed by the signal name (or names) to trap on:

```
trap command signal
```

Remember that the scripts are normally interpreted from top to bottom, so you must specify the `trap` command before the part of the script you wish to protect.

To reset a trap condition to the default, simply specify the command as `-`. To ignore a signal, set the command to the empty string `''`. A `trap` command with no parameters prints out the current list of traps and actions.

The following table lists the more important signals covered by the X/Open standard that can be caught (with the conventional signal number in parentheses). More details can be found in the `signal` manual pages in section 7 of the online manual (`man 7 signal`).

Signal	Description
`HUP` (1)	Hang up; usually sent when a terminal goes offline, or a user logs out
`INT` (2)	Interrupt; usually sent by pressing Ctrl+C
`QUIT` (3)	Quit; usually sent by pressing Ctrl+\
`ABRT` (6)	Abort; usually sent on some serious execution error
`ALRM` (14)	Alarm; usually used for handling timeouts
`TERM` (15)	Terminate; usually sent by the system when it's shutting down

Try It Out — Trapping Signals

The following script demonstrates some simple signal handling:

```
#!/bin/sh

trap 'rm -f /tmp/my_tmp_file_$$' INT
echo creating file /tmp/my_tmp_file_$$
date > /tmp/my_tmp_file_$$

echo "press interrupt (CTRL-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
    echo File exists
    sleep 1
done
echo The file no longer exists

trap INT
echo creating file /tmp/my_tmp_file_$$
date >  /tmp/my_tmp_file_$$

echo "press interrupt (control-C) to interrupt ...."
while [ -f /tmp/my_tmp_file_$$ ]; do
    echo File exists
    sleep 1
done

echo we never get here
exit 0
```

If you run this script, holding down Ctrl and then pressing C (or whatever your interrupt key combination is) in each of the loops, you get the following output:

```
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
The file no longer exists
creating file /tmp/my_tmp_file_141
press interrupt (CTRL-C) to interrupt ....
File exists
File exists
File exists
File exists
```

How It Works

This script uses the `trap` command to arrange for the command `rm -f /tmp/my_tmp_file_$$` to be executed when an `INT` (interrupt) signal occurs. The script then enters a `while` loop that continues while the file exists. When the user presses Ctrl+C, the statement `rm -f /tmp/my_tmp_file_$$` is executed, and then the `while` loop resumes. Since the file has now been deleted, the first `while` loop terminates normally.

The script then uses the `trap` command again, this time to specify that no command be executed when an `INT` signal occurs. It then re-creates the file and loops inside the second `while` statement. When the user presses Ctrl+C this time, no statement is configured to execute, so the default behavior occurs, which is to immediately terminate the script. Because the script terminates immediately, the final `echo` and `exit` statements are never executed.

---

unset

The `unset` command removes variables or functions from the environment. It can't do this to read-only variables defined by the shell itself, such as IFS. It's not often used.

The following script writes `Hello World` once and a newline the second time:

```
#!/bin/sh

foo="Hello World"
echo $foo

unset foo
echo $foo
```

> **Writing `foo=` would have a very similar, but not identical, effect to `unset` in the preceding program. Writing `foo=` has the effect of setting `foo` to `null`, but `foo` still exists. Using `unset foo` has the effect of removing the variable `foo` from the environment.**

Two More Useful Commands and Regular Expressions

Before you see how to put this new knowledge of shell programming to use, let's look at a couple of other very useful commands, which, although not part of the shell, are often useful when writing shell programs. Along the way we will also be looking at regular expressions, a pattern-matching feature that crops up all over Linux and its associated programs.

The find Command

The first command you will look at is `find`. This command, which you use to search for files, is extremely useful, but newcomers to Linux often find it a little tricky to use, not least because it takes options, tests, and action-type arguments, and the results of one argument can affect the processing of subsequent arguments.

Before delving into the options, tests, and arguments, let's look at a very simple example for the file `test` on your local machine. Do this as root to ensure that you have permissions to search the whole machine:

```
# find / -name test -print
/usr/bin/test
#
```

Depending on your installation, you may well find several other files also called `test`. As you can probably guess, this says "search starting at / for a file named *test* and then print out the name of the file." Easy, wasn't it? Of course.

However, it did take quite a while to run on our machine, and the disk on our Windows machine on the network rattled away as well. This is because our Linux machine mounts (using SAMBA) a chunk of the Windows machine's file system. It seems like that might have been searched as well, even though we knew the file we were looking for would be on the Linux machine.

This is where the first of the options comes in. If you specify `-mount`, you can tell `find` not to search mounted directories:

```
# find / -mount -name test -print
/usr/bin/test
#
```

We still find the file on our machine, but faster this time, and without searching other mounted file systems.

The full syntax for the `find` command is as follows:

```
find [path] [options] [tests] [actions]
```

The `path` part is nice and easy: You can use either an absolute path, such as `/bin`, or a relative path, such as `.`. If you need to, you can also specify multiple paths — for example, `find /var /home`.

There are several options; the main ones are shown in the following table:

Option	Meaning
`-depth`	Search the contents of a directory before looking at the directory itself.
`-follow`	Follow symbolic links.
`-maxdepths N`	Search at most *N* levels of the directory when searching.
`-mount` (or `-xdev`)	Don't search directories on other file systems.

Now for the tests. A large number of tests can be given to `find`, and each test returns either `true` or `false`. When `find` is working, it considers each file it finds in turn and applies each test, in the order they were defined, on that file. If a test returns `false`, then `find` stops considering the file it is currently looking at and moves on; if the test returns `true`, then `find` processes the next test or action on the current file. The tests listed in the following table are just the most common; consult the manual pages for the extensive list of possible tests you can apply using `find`.

Test	Meaning
`-atime N`	The file was last accessed *N* days ago.
`-mtime N`	The file was last modified *N* days ago.
`-name pattern`	The name of the file, excluding any path, matches the pattern provided. To ensure that the pattern is passed to `find`, and not evaluated by the shell immediately, the pattern must always be in quotes.
`-newer otherfile`	The file is newer than the file `otherfile`.
`-type C`	The file is of type *C*, where *C* can be of a particular type; the most common are "d" for a directory and "f" for a regular file. For other types consult the manual pages.
`-user username`	The file is owned by the user with the given name.

You can also combine tests using operators. Most have two forms: a short form and a longer form, as shown in the following table:

Operator, Short Form	Operator, Long Form	Meaning
`!`	`-not`	Invert the test.
`-a`	`-and`	Both tests must be true.
`-o`	`-or`	Either test must be true.

You can force the precedence of tests and operators by using parentheses. Since these have a special meaning to the shell, you also have to quote the braces using a backslash. In addition, if you use a pattern for the filename, then you must use quotes so that the name is not expanded by the shell but passed directly to the `find` command. For example, if you wanted to write the test "newer than file X or called a name that starts with an underscore," you could write the following test:

```
\(-newer X -o -name "_*" \)
```

We present an example just after the next "How it Works" section.

Try It Out — Using find with Tests

Try searching in the current directory for files modified more recently than the file `while2`:

```
$ find . -newer while2 -print
.
./elif3
./words.txt
./words2.txt
./_trap
$
```

That looks good, except that you also find the current directory, which you didn't want. You were interested only in regular files, so you add an additional `test`, `-type f`:

```
$ find . -newer while2 -type f -print
./elif3
./words.txt
./words2.txt
./_trap
$
```

How It Works

How did it work? You specified that `find` should search in the current directory (`.`), for files newer than the file `while2` (`-newer while2`) and that, if that test passed, then to also test that the file was a regular file (`-type f`). Finally, you used the action you already met, `-print`, just to confirm which files were found.

Now find files that either start with an underscore or are newer than the file `while2`, but must in either case be regular files. This will show you how to combine tests using parentheses:

```
$ find .  \( -name "_*" -or -newer while2 \) -type f -print
./elif3
./words.txt
./words2.txt
./_break
./_if
./_set
./_shift
./_trap
./_unset
./_until
$
```

That wasn't so hard, was it? You had to escape the parentheses so that they were not processed by the shell, and quote the `*` so that it was passed directly into `find` as well.

Now that you can reliably search for files, look at the actions you can perform when you find a file matching your specification. Again, this is just a list of the most common actions; the manual page has the full set.

Action	Meaning
`-exec command`	Execute a command. This is one of the most common actions. See the explanation following this table for how parameters may be passed to the command. This action must be terminated with a `\;` character pair.
`-ok command`	Like `-exec`, except that it prompts for user confirmation of each file on which it will carry out the command before executing the command. This action must be terminated with a `\;` character pair.
`-print`	Print out the name of the file.
`-ls`	Use the command `ls -dils` on the current file.

The `-exec` and `-ok` commands take subsequent parameters on the line as part of their parameters, until terminated with a `\;` sequence. Effectively, the `-exec` and `-ok` commands are executing an embedded command, so that embedded command has to be terminated with an escaped semicolon so that the `find` command can determine when it should resume looking for command-line options that are intended for itself. The magic string `"{}"` is a special type of parameter to an `-exec` or `-ok` command and is replaced with the full path to the current file.

That explanation is perhaps not so easy to understand, but an example should make things clearer. Take a look at a simple example, using a nice safe command like `ls`:

```
$ find . -newer while2 -type f -exec ls -l {} \;
-rwxr-xr-x    1 rick     rick          275 Feb  8 17:07 ./elif3
-rwxr-xr-x    1 rick     rick          336 Feb  8 16:52 ./words.txt
-rwxr-xr-x    1 rick     rick         1274 Feb  8 16:52 ./words2.txt
-rwxr-xr-x    1 rick     rick          504 Feb  8 18:43 ./_trap
$
```

As you can see, the `find` command is extremely useful; it just takes a little practice to use it well. However, that practice will pay dividends, so do experiment with the `find` command.

The grep Command

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

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

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

If no filenames are given, it searches standard input.

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

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

Try It Out — Basic grep Usage

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

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

How It Works

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

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

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

Regular Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Command Execution

When you're writing scripts, you often need to capture the result of a command's execution for use in the shell script; that is, you want to execute a command and put the output of the command into a variable.

You can do this by using the `$(command)` syntax introduced in the earlier `set` command example. There is also an older form, `` `command` ``, that is still in common usage.

> **Note that with the older form of the command execution, the backtick, or backquote, (`` ` ``), is used, not the single quote (`'`) that we used in earlier shell quoting (to protect against variable expansion). Use this form for shell scripts only when you need them to be very portable.**

All new scripts should use the `$(...)` form, which was introduced to avoid some rather complex rules covering the use of the characters `$`, `` ` ``, and `\` inside the backquoted command. If a backtick is used within the `` `...` `` construct, it must be escaped with a `\` character. These relatively obscure characters often confuse programmers, and sometimes even experienced shell programmers are forced to experiment to get the quoting correct in backticked commands.

The result of the `$(command)` is simply the output from the command. Note that this isn't the return status of the command but the string output, as shown here:

```
#!/bin/sh

echo The current directory is $PWD
echo The current users are $(who)

exit 0
```

Since the current directory is a shell environment variable, the first line doesn't need to use this command execution construct. The result of `who`, however, does need this construct if it is to be available to the script.

If you want to get the result into a variable, you can just assign it in the usual way:

```
whoisthere=$(who)
echo $whoisthere
```

The capability to put the result of a command into a script variable is very powerful, as it makes it easy to use existing commands in scripts and capture their output. If you ever find yourself trying to convert a set of parameters that are the output of a command on standard output and capture them as arguments for a program, you may well find the command `xargs` can do it for you. Look in the manual pages for further details.

A problem sometimes arises when the command you want to invoke outputs some white space before the text you want, or more output than you require. In such a case, you can use the `set` command as shown earlier.

Arithmetic Expansion

We've already used the `expr` command, which enables simple arithmetic commands to be processed, but this is quite slow to execute because a new shell is invoked to process the `expr` command.

A newer and better alternative is `$((…))` expansion. By enclosing the expression you wish to evaluate in `$((…))`, you can perform simple arithmetic much more efficiently:

```
#!/bin/sh

x=0
while [ "$x" -ne 10 ]; do
    echo $x
    x=$(($x+1))
done

exit 0
```

> **Notice that this is subtly different from the `x=$( … )` command. The double parentheses are used for arithmetic substitution. The single parentheses form shown earlier is used for executing commands and grabbing the output.**

Parameter Expansion

You've seen the simplest form of parameter assignment and expansion:

```
foo=fred
echo $foo
```

A problem occurs when you want to append extra characters to the end of a variable. Suppose you want to write a short script to process files called `1_tmp` and `2_tmp`. You could try this:

```
#!/bin/sh

for i in 1 2
do
    my_secret_process $i_tmp
done
```

But on each loop, you'll get the following:

```
my_secret_process: too few arguments
```

What went wrong?

The problem is that the shell tried to substitute the value of the variable `$i_tmp`, which doesn't exist. The shell doesn't consider this an error; it just substitutes nothing, so no parameters at all were passed to `my_secret_process`. To protect the expansion of the `$i` part of the variable, you need to enclose the `i` in `braces` like this:

```
#!/bin/sh

for i in 1 2
do
    my_secret_process ${i}_tmp
done
```

On each loop, the value of `i` is substituted for `${i}` to give the actual filenames. You substitute the value of the parameter into a string.

You can perform many parameter substitutions in the shell. Often, these provide an elegant solution to many parameter-processing problems. The common ones are shown in the following table:

Parameter Expansion	Description
`${param:-default}`	If `param` is null, then set it to the value of `default`.
`${#param}`	Gives the length of `param`
`${param%word}`	From the end, removes the smallest part of `param` that matches `word` and returns the rest
`${param%%word}`	From the end, removes the longest part of `param` that matches `word` and returns the rest
`${param#word}`	From the beginning, removes the smallest part of `param` that matches `word` and returns the rest
`${param##word}`	From the beginning, removes the longest part of `param` that matches `word` and returns the rest

These substitutions are often useful when you're working with strings. The last four, which remove parts of strings, are especially useful for processing filenames and paths, as the following example shows.

Try It Out Parameter Processing

Each portion of the following script illustrates the parameter-matching operators:

```
#!/bin/sh

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

This gives the following output:

```
bar
fud
usr/bin/X11/startx
startx
/usr/local/etc
/usr
```

How It Works

The first statement, `${foo:-bar}`, gives the value `bar`, because `foo` had no value when the statement was executed. The variable `foo` is unchanged, as it remains unset.

> **`${foo:=bar}`, however, would set the variable to `$foo`. This string operator checks that `foo` exists and isn't `null`. If it isn't null, then it returns its value, but otherwise it sets `foo` to `bar` and returns that instead.**
>
> **`${foo:?bar}` will print `foo: bar` and abort the command if `foo` doesn't exist or is set to null. Lastly, `${foo:+bar}` returns `bar` if `foo` exists and isn't `null`. What a set of choices!**

The `{foo#*/}` statement matches and removes only the left `/` (remember `*` matches zero or more characters). The `{foo##*/}` matches and removes as much as possible, so it removes the rightmost `/` and all the characters before it.

The `{bar%local*}` statement matches characters from the right until the first occurrence of `local` (followed by any number of characters) is matched, but the `{bar%%local*}` matches as many characters as possible from the right until it finds the leftmost `local`.

Since both UNIX and Linux are based heavily around the idea of filters, the result of one operation must often be redirected manually. Let's say you want to convert a `GIF` file into a `JPEG` file using the `cjpeg` program:

```
$ cjpeg image.gif > image.jpg
```

Sometimes you may want to perform this type of operation on a large number of files. How do you automate the redirection? It's as easy as this:

```
#!/bin/sh

for image in  *.gif
do
  cjpeg $image > ${image%%gif}jpg
done
```

This script, `giftojpeg`, creates a `JPEG` file for each `GIF` file in the current directory.

Here Documents

One special way of passing input to a command from a shell script is to use a *here document*. This document allows a command to execute as though it were reading from a file or the keyboard, whereas in fact it's getting input from the script.

A here document starts with the leader <<, followed by a special sequence of characters that is repeated at the end of the document. << is the shell's label redirector, which in this case forces the command input to be the here document. This special sequence acts as a marker to tell the shell where the here document ends. The marker sequence must not appear in the lines to be passed to the command, so it's best to make them memorable and fairly unusual.

Try It Out — Using Here Documents

The simplest example is simply to feed input to the `cat` command:

```
#!/bin/sh

cat <<!FUNKY!
hello
this is a here
document
!FUNKY!
```

This gives the following output:

```
hello
this is a here
document
```

Here documents might seem a rather curious feature, but they're very powerful because they enable you to invoke an interactive program such as an editor and feed it some predefined input. However, they're more commonly used for outputting large amounts of text from inside a script, as you saw previously, and avoiding having to use `echo` statements for each line. You can use exclamation marks (`!`) on each side of the identifier to ensure that there's no confusion.

If you wish to process several lines in a file in a predetermined way, you could use the `ed` line editor and feed it commands from a here document in a shell script.

Try It Out — Another Use for a Here Document

1. Start with a file called `a_text_file` that contains the following lines:

```
That is line 1
That is line 2
That is line 3
That is line 4
```

2. You can edit this file using a combination of a here document and the `ed` editor:

```
#!/bin/sh
```

```
ed a_text_file <<!FunkyStuff!
3
d
.,\$s/is/was/
w
q
!FunkyStuff!

exit 0
```

If you run this script, the file now contains the following:

```
That is line 1
That is line 2
That was line 4
```

How It Works

The shell script simply invokes the `ed` editor and passes to it the commands that it needs to move to the third line, delete the line, and then replace it with what was in the current line (because line 3 was deleted, the current line is now what was the last line). These `ed` commands are taken from the lines in the script that form the here document — the lines between the markers `!FunkyStuff!`.

> **Notice the `\` inside the here document to protect the `$` from shell expansion. The `\` escapes the `$`, so the shell knows not to try to expand `$s/is/was/` to its value, which of course it doesn't have. Instead, the shell passes the text `\$` as `$`, which can then be interpreted by the `ed` editor.**

Debugging Scripts

Debugging shell scripts is usually quite easy, but there are no specific tools to help. In this section we'll quickly summarize the common methods.

When an error occurs, the shell will normally print out the line number of the line containing the error. If the error isn't immediately apparent, you can add some extra `echo` statements to display the contents of variables and test code fragments by simply typing them into the shell interactively.

Since scripts are interpreted, there's no compilation overhead in modifying and retrying a script. The main way to trace more complicated errors is to set various shell options. To do this, you can either use command-line options after invoking the shell or use the `set` command. The following table summarizes the options:

Command Line Option	`set` Option	Description
`sh -n <script>`	`set -o noexec` `set -n`	Checks for syntax errors only; doesn't execute commands
`sh -v <script>`	`set -o verbose` `set -v`	Echoes commands before running them
`sh -x <script>`	`set -o xtrace` `set -x`	Echoes commands after processing on the command line
`sh -u <script>`	`set -o nounset` `set -u`	Gives an error message when an undefined variable is used

You can set the `set` option flags on, using `-o`, and off, using `+o`, and likewise for the abbreviated versions. You can achieve a simple execution trace by using the `xtrace` option. For an initial check, you can use the command-line option, but for finer debugging, you can put the `xtrace` flags (setting an execution trace on and off) inside the script around the problem code. The execution trace causes the shell to print each line in the script, with variables expanded, before executing the line.

Use the following command to turn `xtrace` on:

```
set -o xtrace
```

Use this command to turn `xtrace` off again:

```
set +o xtrace
```

The level of expansion is denoted (by default) by the number of + signs at the start of each line. You can change the + to something more meaningful by setting the `PS4` shell variable in your shell configuration file.

In the shell, you can also find out the program state wherever it exits by trapping the `EXIT` signal with a line something like the following placed at the start of the script:

```
trap 'echo Exiting: critical variable = $critical_variable' EXIT
```

Going Graphical — The dialog Utility

Before we finish discussing shell scripts, there is one more feature that, although not strictly part of the shell, is generally useful only from shell programs, so we cover it here.

If you know that your script will only ever need to run on the Linux console, there is a rather neat way to brighten up your scripts using a utility command called `dialog`. This command uses text mode graphics and color, but it still looks pleasantly graphical.