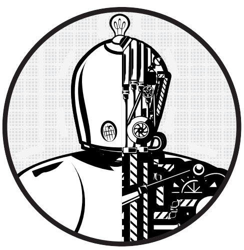
11

INTRODUCTION TO SHELL SCRIPTS



If you can enter commands into the shell, you can write shell scripts (also known as Bourne shell scripts). A *shell script* is a series of commands written in a file; the shell reads the commands from the file just as it would if you typed them into a terminal.

11.1 Shell Script Basics

Bourne shell scripts generally start with the following line, which indicates that the `/bin/sh` program should execute the commands in the script file. (Make sure that no whitespace appears at the beginning of the script file.)

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

The #! part is called a *shebang*; you'll see it in other scripts in this book. You can list any commands that you want the shell to execute following the #!/bin/sh line. For example:

```
#!/bin/sh
#
# Print something, then run ls

echo About to run the ls command.
ls
```

NOTE *A # character at the beginning of a line indicates that the line is a comment; that is, the shell ignores anything on a line after a #. Use comments to explain parts of your scripts that are difficult to understand.*

After creating a shell script and setting its permissions, you can run it by placing the script file in one of the directories in your command path and then running the script name on the command line. You can also run *./script* if the script is located in your current working directory, or you can use the full pathname.

As with any program on Unix systems, you need to set the executable bit for a shell script file, but you must also set the read bit in order for the shell to read the file. The easiest way to do this is as follows:

```
$ chmod +rx script
```

This chmod command allows other users to read and execute *script*. If you don't want that, use the absolute mode 700 instead (and refer to Section 2.17 for a refresher on permissions).

With the basics behind us, let's look at some of the limitations of shell scripts.

11.1.1 Limitations of Shell Scripts

The Bourne shell manipulates commands and files with relative ease. In Section 2.14, you saw the way the shell can redirect output, one of the important elements of shell script programming. However, the shell script is only one tool for Unix programming, and although scripts have considerable power, they also have limitations.

One of the main strengths of shell scripts is that they can simplify and automate tasks that you can otherwise perform at the shell prompt, like manipulating batches of files. But if you're trying to pick apart strings, perform repeated arithmetic computations, or access complex databases, or if you want functions and complex control structures, you're better off using a scripting language like Python, Perl, or awk, or perhaps even a compiled language like C. (This is important, so we'll repeat it throughout the chapter.)

Finally, be aware of your shell script sizes. Keep your shell scripts short. Bourne shell scripts aren't meant to be big (though you will undoubtedly encounter some monstrosities).

11.2 Quoting and Literals

One of the most confusing elements of working with the shell and scripts is when to use quotation marks (or *quotes*) and other punctuation, and why it's sometimes necessary to do so. Let's say you want to print the string `$100` and you do the following:

```
$ echo $100
00
```

Why did this print `00`? Because the shell saw `$1`, which is a shell variable (we'll cover it soon). So you might think that if you surround it with double quotes, the shell will leave the `$1` alone. But it still doesn't work:

```
$ echo "$100"
00
```

Then you ask a friend, who says that you need to use single quotes instead:

```
$ echo '$100'
$100
```

Why did this particular incantation work?

11.2.1 Literals

When you use quotes, you're often trying to create a *literal,* a string that you want the shell to pass to the command line untouched. In addition to the `$` in the example that you just saw, other similar circumstances include when you want to pass a * character to a command such as `grep` instead of having the shell expand it, and when you need to need to use a semicolon (`;`) in a command.

When writing scripts and working on the command line, just remember what happens whenever the shell runs a command:

1. Before running the command, the shell looks for variables, globs, and other substitutions and performs the substitutions if they appear.
2. The shell passes the results of the substitutions to the command.

Problems involving literals can be subtle. Let's say you're looking for all entries in */etc/passwd* that match the regular expression `r.*t` (that is, a line that contains an `r` followed by a `t` later in the line, which would enable you to search for usernames such as `root` and `ruth` and `robot`). You can run this command:

```
$ grep r.*t /etc/passwd
```

It works most of the time, but sometimes it mysteriously fails. Why? The answer is probably in your current directory. If that directory contains files with names such as *r.input* and *r.output*, then the shell expands `r.*t` to `r.input r.output` and creates this command:

```
$ grep r.input r.output /etc/passwd
```

The key to avoiding problems like this is to first recognize the characters that can get you in trouble and then apply the correct kind of quotes to protect the characters.

11.2.2 Single Quotes

The easiest way to create a literal and make the shell leave a string alone is to enclose the entire string in single quotes, as in this example with `grep` and the * character:

```
$ grep 'r.*t' /etc/passwd
```

As far as the shell is concerned, all characters between the two single quotes, including spaces, make up a single parameter. Therefore, the following command does *not* work, because it asks the `grep` command to search for the string `r.*t /etc/passwd` in the standard input (because there's only one parameter to `grep`):

```
$ grep 'r.*t /etc/passwd'
```

When you need to use a literal, you should always turn to single quotes first, because you're guaranteed that the shell won't try *any* substitutions. As a result, it's a generally clean syntax. However, sometimes you need a little more flexibility, so you can turn to double quotes.

11.2.3 Double Quotes

Double quotes (") work just like single quotes, except that the shell expands any variables that appear within double quotes. You can see the difference by running the following command and then replacing the double quotes with single quotes and running it again.

```
$ echo "There is no * in my path: $PATH"
```

When you run the command, notice that the shell substitutes for `$PATH` but does not substitute for the *.

NOTE *If you're using double quotes when printing large amounts of text, consider using a* here *document, as described in Section 11.9.*

11.2.4 Passing a Literal Single Quote

One tricky part to using literals with the Bourne shell comes when passing a literal single quote to a command. One way to do this is to place a backslash before the single quote character:

```
$ echo I don\'t like contractions inside shell scripts.
```

The backslash and quote *must* appear outside any pair of single quotes, and a string such as `'don\'t` results in a syntax error. Oddly enough, you can enclose the single quote inside double quotes, as shown in the following example (the output is identical to that of the preceding command):

```
$ echo "I don't like contractions inside shell scripts."
```

If you're in a bind and you need a general rule to quote an entire string with no substitutions, follow this procedure:

1. Change all instances of `'` (single quote) to `'\''` (single quote, backslash, single quote, single quote).
2. Enclose the entire string in single quotes.

Therefore, you can quote an awkward string such as `this isn't a forward slash: \` as follows:

```
$ echo 'this isn'\''t a forward slash: \'
```

NOTE *It's worth repeating that when you quote a string, the shell treats everything inside the quotes as a single parameter. Therefore,* `a b c` *counts as three parameters, but* `a "b c"` *is only two.*

11.3 Special Variables

Most shell scripts understand command-line parameters and interact with the commands that they run. To take your scripts from being just a simple list of commands to becoming more flexible shell script programs, you need to know how to use the special Bourne shell variables. These special variables are like any other shell variable as described in Section 2.8, except that you cannot change the values of certain ones.

NOTE *After reading the next few sections, you'll understand why shell scripts accumulate many special characters as they are written. If you're trying to understand a shell script and you come across a line that looks completely incomprehensible, pick it apart piece by piece.*

11.3.1 Individual Arguments: $1, $2, ...

`$1`, `$2`, and all variables named as positive nonzero integers contain the values of the script parameters, or arguments. For example, say the name of the following script is *pshow*:

```
#!/bin/sh
echo First argument: $1
echo Third argument: $3
```

Try running the script as follows to see how it prints the arguments:

```
$ ./pshow one two three
First argument: one
Third argument: three
```

The built-in shell command `shift` can be used with argument variables to remove the first argument (`$1`) and advance the rest of the arguments forward. Specifically, `$2` becomes `$1`, `$3` becomes `$2`, and so on. For example, assume that the name of the following script is *shiftex*:

```
#!/bin/sh
echo Argument: $1
shift
echo Argument: $1
shift
echo Argument: $1
```

Run it like this to see it work:

```
$ ./shiftex one two three
Argument: one
Argument: two
Argument: three
```

As you can see, `shiftex` prints all three arguments by printing the first, shifting the remaining arguments, and repeating.

11.3.2 Number of Arguments: $#

The `$#` variable holds the number of arguments passed to a script and is especially important when running `shift` in a loop to pick through arguments. When `$#` is `0`, no arguments remain, so `$1` is empty. (See Section 11.6 for a description of loops.)

11.3.3 All Arguments: $@

The `$@` variable represents all of a script's arguments, and it is very useful for passing them to a command inside the script. For example, Ghostscript commands (`gs`) are usually long and complicated. Suppose you want a shortcut for rasterizing a PostScript file at 150 dpi, using the standard output stream, while also leaving the door open for passing other options to gs. You could write a script like this to allow for additional command-line options:

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER -sOutputFile=- -sDEVICE=pnmraw $@
```

NOTE *If a line in your shell script gets too long for your text editor, you can split it up with a backslash (\). For example, you can alter the preceding script as follows:*

```
#!/bin/sh
gs -q -dBATCH -dNOPAUSE -dSAFER \
   -sOutputFile=- -sDEVICE=pnmraw $@
```

11.3.4 Script Name: $0

The `$0` variable holds the name of the script, and it is useful for generating diagnostic messages. For example, say your script needs to report an invalid argument that is stored in the `$BADPARM` variable. You can print the diagnostic message with the following line so that the script name appears in the error message:

```
echo $0: bad option $BADPARM
```

All diagnostic error messages should go to the standard error. Recall from Section 2.14.1 that `2>&1` redirects the standard error to the standard output. For writing to the standard error, you can reverse the process with `1>&2`. To do this for the preceding example, use this:

```
echo $0: bad option $BADPARM 1>&2
```

11.3.5 Process ID: $$

The `$$` variable holds the process ID of the shell.

11.3.6 Exit Code: $?

The `$?` variable holds the exit code of the last command that the shell executed. Exit codes, which are critical to mastering shell scripts, are discussed next.

11.4 Exit Codes

When a Unix program finishes, it leaves an *exit code* for the parent process that started the program. The exit code is a number and is sometimes called an *error code* or *exit value*. When the exit code is zero (0), it typically means that the program ran without a problem. However, if the program has an error, it usually exits with a number other than 0 (but not always, as you'll see next).

The shell holds the exit code of the last command in the `$?` special variable, so you can check it out at your shell prompt:

```
$ ls / > /dev/null
$ echo $?
0
$ ls /asdfasdf > /dev/null
ls: /asdfasdf: No such file or directory
$ echo $?
1
```

You can see that the successful command returned 0 and the unsuccessful command returned 1 (assuming, of course, that you don't have a directory named */asdfasdf* on your system).

If you intend to use the exit code of a command, you *must* use or store the code immediately after running the command. For example, if you run `echo $?` twice in a row, the output of the second command is always 0 because the first `echo` command completes successfully.

When writing shell code that aborts a script abnormally, use something like `exit 1` to pass an exit code of 1 back to whatever parent process ran the script. (You may want to use different numbers for different conditions.)

One thing to note is that some programs like `diff` and `grep` use nonzero exit codes to indicate normal conditions. For example, `grep` returns 0 if it finds something matching a pattern and 1 if it doesn't. For these programs, an exit code of 1 is not an error; `grep` and `diff` use the exit code 2 for real problems. If you think a program is using a nonzero exit code to indicate success, read its manual page. The exit codes are usually explained in the EXIT VALUE or DIAGNOSTICS section.

11.5 Conditionals

The Bourne shell has special constructs for conditionals, such as `if`/`then`/`else` and `case` statements. For example, this simple script with an `if` conditional checks to see whether the script's first argument is `hi`:

```
#!/bin/sh
if [ $1 = hi ]; then
   echo 'The first argument was "hi"'
else
   echo -n 'The first argument was not "hi" -- '
   echo It was '"'$1'"'
fi
```

The words `if`, `then`, `else`, and `fi` in the preceding script are shell keywords; everything else is a command. This distinction is extremely important because one of the commands is `[ $1 = "hi" ]` and the `[` character is an actual program on a Unix system, *not* special shell syntax. (This is actually not quite true, as you'll soon learn, but treat it as a separate command in your head for now.) All Unix systems have a command called `[` that performs tests for shell script conditionals. This program is also known as `test` and careful examination of `[` and `test` should reveal that they share an inode, or that one is a symbolic link to the other.

Understanding the exit codes in Section 11.4 is vital, because this is how the whole process works:

1. The shell runs the command after the `if` keyword and collects the exit code of that command.
2. If the exit code is 0, the shell executes the commands that follow the `then` keyword, stopping when it reaches an `else` or `fi` keyword.
3. If the exit code is not 0 and there is an `else` clause, the shell runs the commands after the `else` keyword.
4. The conditional ends at `fi`.

11.5.1 Getting Around Empty Parameter Lists

There is a slight problem with the conditional in the preceding example due to a very common mistake: `$1` could be empty, because the user might not enter a parameter. Without a parameter, the test reads `[ = hi ]`, and the `[` command aborts with an error. You can fix this by enclosing the parameter in quotes in one of two ways (both of which are common):

```
if [ "$1" = hi ]; then
if [ x"$1" = x"hi" ]; then
```

11.5.2 Using Other Commands for Tests

The stuff following `if` is always a command. Therefore, if you want to put the `then` keyword on the same line, you need a semicolon (`;`) after the test command. If you skip the semicolon, the shell passes `then` as a parameter to the test command. (If you don't like the semicolon, you can put the `then` keyword on a separate line.)

There are many possibilities for using other commands instead of the `[` command. Here's an example that uses `grep`:

```
#!/bin/sh
if grep -q daemon /etc/passwd; then
    echo The daemon user is in the passwd file.
else
    echo There is a big problem. daemon is not in the passwd file.
fi
```

11.5.3 elif

There is also an `elif` keyword that lets you string `if` conditionals together, as shown below. But don't get too carried away with `elif`, because the `case` construct that you'll see in Section 11.5.6 is often more appropriate.

```
#!/bin/sh
if [ "$1" = "hi" ]; then
   echo 'The first argument was "hi"'
elif [ "$2" = "bye" ]; then
   echo 'The second argument was "bye"'
else
   echo -n 'The first argument was not "hi" and the second was not "bye"-- '
   echo They were '"'$1'"' and '"'$2'"'
fi
```

11.5.4 && and || Logical Constructs

There are two quick one-line conditional constructs that you may see from time to time: `&&` ("and") and `||` ("or"). The `&&` construct works like this:

```
command1 && command2
```

Here, the shell runs *command1*, and if the exit code is 0, the shell also runs *command2*. The `||` construct is similar; if the command before a `||` returns a nonzero exit code, the shell runs the second command.

The constructs `&&` and `||` often find their way into use in `if` tests, and in both cases, the exit code of the last command run determines how the shell processes the conditional. In the case of the `&&` construct, if the first command fails, the shell uses its exit code for the `if` statement, but if the first command succeeds, the shell uses the exit code of the second command for the conditional. In the case of the `||` construct, the shell uses the exit code of the first command if successful, or the exit code of the second if the first is unsuccessful.

For example:

```
#!/bin/sh
if [ "$1" = hi ] || [ "$1" = bye ]; then
    echo 'The first argument was "'$1'"'
fi
```

If your conditionals include the test (`[`) command, as shown here, you can use `-a` and `-o` instead of `&&` and `||`, as described in the next section.

11.5.5 Testing Conditions

You've seen how [works: The exit code is 0 if the test is true and nonzero when the test fails. You also know how to test string equality with `[ str1 = str2 ]`. However, remember that shell scripts are well suited to operations on entire files because the most useful [tests involve file properties. For example, the following line checks whether *file* is a regular file (not a directory or special file):

```
[ -f file ]
```

In a script, you might see the -f test in a loop similar to this next one, which tests all of the items in the current working directory (you'll learn more about loops in general shortly):

```
for filename in *; do
    if [ -f $filename ]; then
        ls -l $filename
        file $filename
    else
        echo $filename is not a regular file.
    fi
done
```

You can invert a test by placing the ! operator before the test arguments. For example, `[ ! -f file ]` returns true if *file* is not a regular file. Furthermore, the -a and -o flags are the logical "and" and "or" operators (for example, `[ -f file1 -a file2 ]`).

NOTE *Because the* `test` *command is so widely used in scripts, many versions of the Bourne shell (including* `bash`*) incorporate the* `test` *command as a built-in. This can speed up scripts because the shell doesn't have to run a separate command for each test.*

There are dozens of test operations, all of which fall into three general categories: file tests, string tests, and arithmetic tests. The info manual contains complete online documentation, but the test(1) manual page is a fast reference. The following sections outline the main tests. (I've omitted some of the less common ones.)

File Tests

Most file tests, like -f, are called *unary* operations because they require only one argument: the file to test. For example, here are two important file tests:

-e Returns true if a file exists

-s Returns true if a file is not empty

Several operations inspect a file's type, meaning that they can determine whether something is a regular file, a directory, or some kind of special device, as listed in Table 11-1. There are also a number of unary operations that check a file's permissions, as listed in Table 11-2. (See Section 2.17 for an overview of permissions.)

Table 11-1: File Type Operators

Operator	Tests For
-f	Regular file
-d	Directory
-h	Symbolic link
-b	Block device
-c	Character device
-p	Named pipe
-S	Socket

NOTE *The `test` command follows symbolic links (except for the `-h` test). That is, if `link` is a symbolic link to a regular file, `[ -f link ]` returns an exit code of true (0).*

Table 11-2: File Permissions Operators

Operator	Operator
-r	Readable
-w	Writable
-x	Executable
-u	Setuid
-g	Setgid
-k	"Sticky"

Finally, three *binary* operators (tests that need two files as arguments) are used in file tests, but they're not terribly common. Consider this command that includes `-nt` (newer than):

```
[ file1 -nt file2 ]
```

This exits true if *`file1`* has a newer modification date than *`file2`*. The `-ot` (older than) operator does the opposite. And if you need to detect identical hard links, `-ef` compares two files and returns true if they share inode numbers and devices.

String Tests

You've seen the binary string operator `=` that returns true if its operands are equal. The `!=` operator returns true if its operands are not equal. And there are two unary string operations:

`-z` Returns true if its argument is empty (`[ -z "" ]` returns 0)

`-n` Returns true if its argument is not empty (`[ -n "" ]` returns 1)

Arithmetic Tests

It's important to recognize that the equal sign (`=`) looks for *string* equality, not *numeric* equality. Therefore, `[ 1 = 1 ]` returns 0 (true), but `[ 01 = 1 ]` returns false. When working with numbers, use `-eq` instead of the equal sign: `[ 01 -eq 1 ]` returns true. Table 11-3 provides the full list of numeric comparison operators.

Table 11-3: Arithmetic Comparison Operators

Operator	Returns True When the First Argument Is . . . the Second
`-eq`	Equal to
`-ne`	Not equal to
`-lt`	Less than
`-gt`	Greater than
`-le`	Less than or equal to
`-ge`	Greater than or equal to

11.5.6 Matching Strings with case

The `case` keyword forms another conditional construct that is exceptionally useful for matching strings. The `case` conditional does not execute any test commands and therefore does not evaluate exit codes. However, it can do pattern matching. This example should tell most of the story:

```
#!/bin/sh
case $1 in
    bye)
        echo Fine, bye.
        ;;
    hi|hello)
        echo Nice to see you.
        ;;
    what*)
        echo Whatever.
        ;;
    *)
        echo 'Huh?'
        ;;
esac
```

The shell executes this as follows:

1. The script matches `$1` against each case value demarcated with the `)` character.
2. If a case value matches `$1`, the shell executes the commands below the case until it encounters `;;`, at which point it skips to the `esac` keyword.
3. The conditional ends with `esac`.

For each case value, you can match a single string (like `bye` in the preceding example) or multiple strings with `|` (`hi|hello` returns true if `$1` equals `hi` or `hello`), or you can use the `*` or `?` patterns (`what*`). To make a default case that catches all possible values other than the case values specified, use a single `*` as shown by the final case in the preceding example.

NOTE *Each case must end with a double semicolon (`;;`) or you risk a syntax error.*

11.6 Loops

There are two kinds of loops in the Bourne shell: `for` and `while` loops.

11.6.1 for Loops

The `for` loop (which is a "for each" loop) is the most common. Here's an example:

```
#!/bin/sh
for str in one two three four; do
    echo $str
done
```

In this listing, `for`, `in`, `do`, and `done` are all shell keywords. The shell does the following:

1. Sets the variable `str` to the first of the four space-delimited values following the `in` keyword (`one`).
2. Runs the `echo` command between the `do` and `done`.
3. Goes back to the `for` line, setting `str` to the next value (`two`), runs the commands between `do` and `done`, and repeats the process until it's through with the values following the `in` keyword.

The output of this script looks like this:

```
one
two
three
four
```

11.6.2 while Loops

The Bourne shell's `while` loop uses exit codes, like the `if` conditional. For example, this script does 10 iterations:

```
#!/bin/sh
FILE=/tmp/whiletest.$$;
echo firstline > $FILE
```

```
while tail -10 $FILE | grep -q firstline; do
    # add lines to $FILE until tail -10 $FILE no longer prints "firstline"
    echo -n Number of lines in $FILE:' '
    wc -l $FILE | awk '{print $1}'
    echo newline >> $FILE
done

rm -f $FILE
```

Here, the exit code of `grep -q firstline` is the test. As soon as the exit code is nonzero (in this case, when the string `firstline` no longer appears in the last 10 lines in `$FILE`), the loop exits.

You can break out of a `while` loop with the `break` statement. The Bourne shell also has an `until` loop that works just like `while`, except that it breaks the loop when it encounters a zero exit code rather than a nonzero exit code. This said, you shouldn't need to use the `while` and `until` loops very often. In fact, if you find that you need to use `while`, you should probably be using a language like awk or Python instead.

11.7 Command Substitution

The Bourne shell can redirect a command's standard output back to the shell's own command line. That is, you can use a command's output as an argument to another command, or you can store the command output in a shell variable by enclosing a command in `$()`.

This example stores a command inside the `FLAGS` variable. The bold in the second line shows the command substitution.

```
#!/bin/sh
FLAGS=$(grep ^flags /proc/cpuinfo | sed 's/.*://' | head -1)
echo Your processor supports:
for f in $FLAGS; do
    case $f in
        fpu)    MSG="floating point unit"
                ;;
        3dnow)  MSG="3DNOW graphics extensions"
                ;;
        mtrr)   MSG="memory type range register"
                ;;
        *)      MSG="unknown"
                ;;
    esac
    echo $f: $MSG
done
```

This example is somewhat complicated because it demonstrates that you can use both single quotes and pipelines inside the command substitution. The result of the `grep` command is sent to the `sed` command (more about `sed` in Section 11.10.3), which removes anything matching the expression `.*:`, and the result of `sed` is passed to `head`.

It's easy to go overboard with command substitution. For example, don't use `$(ls)` in a script because using the shell to expand `*` is faster. Also, if you want to invoke a command on several filenames that you get as a result of a `find` command, consider using a pipeline to `xargs` rather than command substitution, or use the `-exec` option (see Section 11.10.4).

NOTE *The traditional syntax for command substitution is to enclose the command in backticks (``), and you'll see this in many shell scripts. The `$()` syntax is a newer form, but it is a POSIX standard and is generally easier to read and write.*

11.8 Temporary File Management

It's sometimes necessary to create a temporary file to collect output for use by a later command. When making such a file, make sure that the filename is distinct enough that no other programs will accidentally write to it.

Here's how to use the `mktemp` command to create temporary filenames. This script shows you the device interrupts that have occurred in the last two seconds:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)

cat /proc/interrupts > $TMPFILE1
sleep 2
cat /proc/interrupts > $TMPFILE2
diff $TMPFILE1 $TMPFILE2
rm -f $TMPFILE1 $TMPFILE2
```

The argument to `mktemp` is a template. The `mktemp` command converts the `XXXXXX` to a unique set of characters and creates an empty file with that name. Notice that this script uses variable names to store the filenames so that you only have to change one line if you want to change a filename.

NOTE *Not all Unix flavors come with `mktemp`. If you're having portability problems, it's best to install the GNU coreutils package for your operating system.*

Another problem with scripts that employ temporary files is that if the script is aborted, the temporary files could be left behind. In the preceding example, pressing CTRL-C before the second `cat` command leaves a temporary file in */tmp*. Avoid this if possible. Instead, use the `trap` command to create a signal handler to catch the signal that CTRL-C generates and remove the temporary files, as in this handler:

```
#!/bin/sh
TMPFILE1=$(mktemp /tmp/im1.XXXXXX)
TMPFILE2=$(mktemp /tmp/im2.XXXXXX)
trap "rm -f $TMPFILE1 $TMPFILE2; exit 1" INT
 --snip--
```

You must use exit in the handler to explicitly end script execution, or the shell will continue running as usual after running the signal handler.

NOTE *You don't need to supply an argument to* `mktemp`*; if you don't, the template will begin with a* `/tmp/tmp.` *prefix.*

11.9 Here Documents

Say you want to print a large section of text or feed a lot of text to another command. Rather than use several `echo` commands, you can use the shell's *here document* feature, as shown in the following script:

```
#!/bin/sh
DATE=$(date)
cat <<EOF
Date: $DATE

The output above is from the Unix date command.
It's not a very interesting command.
EOF
```

The items in bold control the here document. The `<<EOF` tells the shell to redirect all lines that follow the standard input of the command that precedes `<<EOF`, which in this case is `cat`. The redirection stops as soon as the `EOF` marker occurs on a line by itself. The marker can actually be any string, but remember to use the same marker at the beginning and end of the here document. Also, convention dictates that the marker be in all uppercase letters.

Notice the shell variable `$DATE` in the here document. The shell expands shell variables inside here documents, which is especially useful when you're printing out reports that contain many variables.

11.10 Important Shell Script Utilities

Several programs are particularly useful in shell scripts. Certain utilities such as `basename` are really only practical when used with other programs, and therefore don't often find a place outside shell scripts. However, others such as `awk` can be quite useful on the command line, too.

11.10.1 basename

If you need to strip the extension from a filename or get rid of the directories in a full pathname, use the `basename` command. Try these examples on the command line to see how the command works:

```
$ basename example.html .html
$ basename /usr/local/bin/example
```

In both cases, `basename` returns `example`. The first command strips the *.html* suffix from *example.html*, and the second removes the directories from the full pathname.

This example shows how you can use `basename` in a script to convert GIF image files to the PNG format:

```
#!/bin/sh
for file in *.gif; do
    # exit if there are no files
    if [ ! -f $file ]; then
        exit
    fi
    b=$(basename $file .gif)
    echo Converting $b.gif to $b.png...
    giftopnm $b.gif | pnmtopng > $b.png
done
```

11.10.2 awk

The `awk` command is not a simple single-purpose command; it's actually a powerful programming language. Unfortunately, `awk` usage is now something of a lost art, having been replaced by larger languages such as Python.

The are entire books on the subject of `awk`, including *The AWK Programming Language* by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger (Addison-Wesley, 1988). This said, many, many people use `awk` to do one thing—to pick a single field out of an input stream like this:

```
$ ls -l | awk '{print $5}'
```

This command prints the fifth field of the `ls` output (the file size). The result is a list of file sizes.

11.10.3 sed

The `sed` program (`sed` stands for stream editor) is an automatic text editor that takes an input stream (a file or the standard input), alters it according to some expression, and prints the results to standard output. In many respects, `sed` is like `ed`, the original Unix text editor. It has dozens of operations, matching tools, and addressing capabilities. As with `awk`, entire books have been written about `sed` including a quick reference covering both, *sed & awk Pocket Reference*, 2nd edition, by Arnold Robbins (O'Reilly, 2002).

Although `sed` is a big program, and an in-depth analysis is beyond the scope of this book, it's easy to see how it works. In general, `sed` takes an address and an operation as one argument. The address is a set of lines, and the command determines what to do with the lines.

A very common task for sed is to substitute some text for a regular expression (see Section 2.5.1), like this:

```
$ sed 's/exp/text/'
```

So if you wanted to replace the first colon in */etc/passwd* with a % and send the result to the standard output, you'd do it like this:

```
$ sed 's/:/%/' /etc/passwd
```

To substitute *all* colons in */etc/passwd*, add a g modifier to the end of the operation, like this:

```
$ sed 's/:/%/g' /etc/passwd
```

Here's a command that operates on a per-line basis; it reads */etc/passwd* and deletes lines three through six and sends the result to the standard output:

```
$ sed 3,6d /etc/passwd
```

In this example, 3,6 is the address (a range of lines), and d is the operation (delete). If you omit the address, sed operates on all lines in its input stream. The two most common sed operations are probably s (search and replace) and d.

You can also use a regular expression as the address. This command deletes any line that matches the regular expression *exp*:

```
$ sed '/exp/d'
```

11.10.4 xargs

When you have to run one command on a huge number of files, the command or shell may respond that it can't fit all of the arguments in its buffer. Use xargs to get around this problem by running a command on each filename in its standard input stream.

Many people use xargs with the find command. For example, the script below can help you verify that every file in the current directory tree that ends with *.gif* is actually a GIF (Graphic Interchange Format) image:

```
$ find . -name '*.gif' -print | xargs file
```

In the example above, xargs runs the file command. However, this invocation can cause errors or leave your system open to security problems, because filenames can include spaces and newlines. When writing a script, use the following form instead, which changes the find output separator and the xargs argument delimiter from a newline to a NULL character:

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

xargs starts a *lot* of processes, so don't expect great performance if you have a large list of files.

You may need to add two dashes (--) to the end of your xargs command if there is a chance that any of the target files start with a single dash (-). The double dash (--) can be used to tell a program that any arguments that follow the double dash are filenames, not options. However, keep in mind that not all programs support the use of a double dash.

There's an alternative to xargs when using find: the -exec option. However, the syntax is somewhat tricky because you need to supply a {} to substitute the filename and a literal ; to indicate the end of the command. Here's how to perform the preceding task using only find:

```
$ find . -name '*.gif' -exec file {} \;
```

11.10.5 expr

If you need to use arithmetic operations in your shell scripts, the expr command can help (and even do some string operations). For example, the command expr 1 + 2 prints 3. (Run expr --help for a full list of operations.)

The expr command is a clumsy, slow way of doing math. If you find yourself using it frequently, you should probably be using a language like Python instead of a shell script.

11.10.6 exec

The exec command is a built-in shell feature that replaces the current shell process with the program you name after exec. It carries out the exec() system call that you learned about in Chapter 1. This feature is designed for saving system resources, but remember that there's no return; when you run exec in a shell script, the script and shell running the script are gone, replaced by the new command.

To test this in a shell window, try running exec cat. After you press CTRL-D or CTRL-C to terminate the cat program, your window should disappear because its child process no longer exists.

11.11 Subshells

Say you need to alter the environment in a shell slightly but don't want a permanent change. You can change and restore a part of the environment (such as the path or working directory) using shell variables, but that's a clumsy way to go about things. The easy way around these kinds of problems is to use a *subshell*, an entirely new shell process that you can create just to run a command or two. The new shell has a copy of the original shell's environment, and when the new shell exits, any changes you made to its shell environment disappear, leaving the initial shell to run as normal.

To use a subshell, put the commands to be executed by the subshell in parentheses. For example, the following line executes the command uglyprogram in *uglydir* and leaves the original shell intact:

```
$ (cd uglydir; uglyprogram)
```

This example shows how to add a component to the path that might cause problems as a permanent change:

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

Using a subshell to make a single-use alteration to an environment variable is such a common task that there is even a built-in syntax that avoids the subshell:

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

Pipes and background processes work with subshells, too. The following example uses `tar` to archive the entire directory tree within *orig* and then unpacks the archive into the new directory *target*, which effectively duplicates the files and folders in *orig* (this is useful because it preserves ownership and permissions, and it's generally faster than using a command such as `cp -r`):

```
$ tar cf - orig | (cd target; tar xvf -)
```

WARNING *Double-check this sort of command before you run it to make sure that the* target *directory exists and is completely separate from the* orig *directory.*

11.12 Including Other Files in Scripts

If you need to include another file in your shell script, use the dot (`.`) operator. For example, this runs the commands in the file *config.sh*:

```
. config.sh
```

This "include" file syntax does not start a subshell, and it can be useful for a group of scripts that need to use a single configuration file.

11.13 Reading User Input

The `read` command reads a line of text from the standard input and stores the text in a variable. For example, the following command stores the input in `$var`:

```
$ read var
```

This is a built-in shell command that can be useful in conjunction with other shell features not mentioned in this book.

11.14 When (Not) to Use Shell Scripts

The shell is so feature-rich that it's difficult to condense its important elements into a single chapter. If you're interested in what else the shell can do, have a look at some of the books on shell programming, such as *Unix Shell Programming*, 3rd edition, by Stephen G. Kochan and Patrick Wood (SAMS Publishing, 2003), or the shell script discussion in *The UNIX Programming Environment* by Bran W. Kernighan and Rob Pike (Prentice Hall, 1984).

However, at a certain point (especially when you start using the `read` built-in), you have to ask yourself if you're still using the right tool for the job. Remember what shell scripts do best: manipulate simple files and commands. As stated earlier, if you find yourself writing something that looks convoluted, especially if it involves complicated string or arithmetic operations, you should probably look to a scripting language like Python, Perl, or awk.