7

SEEING THE WORLD AS THE SHELL SEES IT

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

echo Display a line of text

Expansion

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

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

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

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

So what just happened? Why didn't echo print *? As we recall from our work with wildcards, the * character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the * into something else (in this instance, the names of the files in the current working directory) before the echo command is executed. When the ENTER key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the echo command never saw the *, only its expanded result. Knowing this, we can see that echo behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

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

we could carry out the following expansions:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and this:

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even this:

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory, we could do this:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

PATHNAME EXPANSION OF HIDDEN FILES

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as the following does not reveal hidden files:

```
echo *
```

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

```
echo .*
```

It almost works. However, if we examine the results closely, we will see that the names . and .. will also appear in the results. Because these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the following command:

```
ls -d .* | less
```

To better perform pathname expansion in this situation, we have to employ a more specific pattern.

```
echo .[!.]*
```

This pattern expands into every filename that begins with only one period followed by any other characters. This will work correctly with most hidden files (though it still won't include filenames with multiple leading periods). The `ls` command with the `-A` option ("almost all") will provide a correct listing of hidden files.

```
ls -A
```

Tilde Expansion

As we may recall from our introduction to the `cd` command, the tilde character (~) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user.

```
[me@linuxbox ~]$ echo ~
/home/me
```

If user "foo" has an account, then it expands into this:

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator.

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the following form:

```
$((expression))
```

where *expression* is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion supports only integers (whole numbers, no decimals) but can perform quite a number of different operations. Table 7-1 describes a few of the supported operators.

Table 7-1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion supports only integer arithmetic, results are integers)
%	Modulo, which simply means “remainder”
**	Exponentiation

Spaces are not significant in arithmetic expressions, and expressions may be nested. For example, to multiply 5 squared by 3, we can use this:

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

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the previous example and get the same result using a single expansion instead of two.

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

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

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

Arithmetic expansion is covered in greater detail in Chapter 34.

Brace Expansion

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

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

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

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

In bash version 4.0 and newer, integers may also be *zero-padded* like so:

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

Here is a range of letters in reverse order:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested.

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

So, what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric "Year-Month" format. This way, the directory names would sort in chronological order. We could type out a complete list of directories, but that's a lot of work, and it's error-prone. Instead, we could do this:

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
2007-01  2007-07  2008-01  2008-07  2009-01  2009-07
2007-02  2007-08  2008-02  2008-08  2009-02  2009-08
2007-03  2007-09  2008-03  2008-09  2009-03  2009-09
2007-04  2007-10  2008-04  2008-10  2009-04  2009-10
2007-05  2007-11  2008-05  2008-11  2009-05  2009-11
2007-06  2007-12  2008-06  2008-12  2009-06  2009-12
```

Pretty slick!

Parameter Expansion

We're going to touch only briefly on parameter expansion in this chapter, but we'll be covering it extensively later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to

give each chunk a name. Many such chunks, more properly called *variables*, are available for your examination. For example, the variable named USER contains your username. To invoke parameter expansion and reveal the contents of USER, you would do this:

```
[me@linuxbox ~]$ echo $USER
me
```

To see a list of available variables, try this:

```
[me@linuxbox ~]$ printenv | less
```

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place, and the echo command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place but will result in an empty string.

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion.

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

One of my favorites goes something like this:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of `which cp` as an argument to the `ls` command, thereby getting the listing of the `cp` program without having to

know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output is shown here).

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/gpg-zip:      Bourne shell script text executable
/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'
/usr/bin/gzip:         symbolic link to `../../bin/gzip'
/usr/bin/mzip:         symbolic link to `mtools'
```

In this example, the results of the pipeline became the argument list of the `file` command.

There is an alternate syntax for command substitution in older shell programs that is also supported in `bash`. It uses *backquotes* instead of the dollar sign and parentheses.

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take, for example, the following:

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

or this one:

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, *word splitting* by the shell removed extra whitespace from the `echo` command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of `$1` because it was an undefined variable. The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is *double quotes*. If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are `$` (dollar sign), `\` (backslash), and `` ` `` (backtick). This means that word splitting, pathname expansion, tilde expansion, and brace expansion are suppressed; however, parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called `two words.txt`. If we tried to use this on the command line, word splitting would cause this to be treated as two separate arguments rather than the desired single argument.

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we stop the word splitting and get the desired result; further, we can even repair the damage.

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me   me   18 2018-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes.

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes.

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4 February 2020
Su Mo Tu We Th Fr Sa
                1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word splitting appears to remove extra spaces in our text.

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

By default, word splitting looks for the presence of spaces, tabs, and newlines (line feed characters) and treats them as *delimiters* between words. This means unquoted spaces, tabs, and newlines are not considered to be part of the text. They serve only as separators. Because they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

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

then word splitting is suppressed and the embedded spaces are not treated as delimiters; rather, they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox ~]$ echo $(cal)
February 2020 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17
18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
   February 2020
Su Mo Tu We Th Fr Sa
                 1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments. In the second, it resulted in a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

If we need to suppress *all* expansions, we use *single quotes*. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes we want to quote only a single character. To do this, we can precede a character with a backslash, which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion.

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include `$`, `!`, `&`, spaces, and others. To include a special character in a filename, we can do this:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing `\\`. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

Backslash Escape Sequences

In addition to its role as the escape character, the backslash is used as part of a notation to represent certain special characters called *control codes*. The first 32 characters in the ASCII coding scheme are used to transmit commands to Teletype-like devices. Some of these codes are familiar (tab, backspace, line feed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).

Table 7-2 lists some of the common backslash escape sequences.

Table 7-2: Backslash Escape Sequences

Escape sequence	Meaning
`\a`	Bell (an alert that causes the computer to beep)
`\b`	Backspace
`\n`	Newline; on Unix-like systems, this produces a line feed
`\r`	Carriage return
`\t`	Tab

The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the shell.

Adding the `-e` option to `echo` will enable interpretation of escape sequences. You can also place them inside `$' '`. Here, using the `sleep` command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:

```
sleep 10; echo -e "Time's up\a"
```

We could also do this:

```
sleep 10; echo "Time's up" $'\a'
```

Summing Up

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they work. In fact, it could be argued that they are the most important subjects to learn about the shell. Without a proper understanding of expansion, the shell will always be a

source of mystery and confusion, with much of its potential power wasted.