Expansion

Each time we type a command line and press the enter key, bash performs several processes 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 type something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the **echo** command. **echo** is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

```
[me@linuxbox me]$ 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 me]$ 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 our earlier lessons, we will see that they are really expansions. Given a home directory that looks like this:

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

we could carry out the following expansions:

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

and:

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

or even:

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

and looking beyond our home directory:

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

Tilde Expansion

As we 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 me]$ echo ~
/home/me
```

If user “foo” has an account, then:

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

Arithmetic Expansion

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

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

Arithmetic expansion uses the form:

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

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

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations.

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply five squared by three:

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

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

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

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

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

Brace Expansion

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

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

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

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

A range of letters in reverse order:

```
[me@linuxbox me]$ 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 me]$ 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 a photographer and had a large collection of images 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 will 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 too. Instead, we could do this:

```
[me@linuxbox me]$ mkdir Photos
[me@linuxbox me]$ cd Photos
[me@linuxbox Photos]$ mkdir {2017..2019}-{01..12}
[me@linuxbox Photos]$ ls
2017-01 2017-07 2018-01 2018-07 2019-01 2019-07
2017-02 2017-08 2018-02 2018-08 2019-02 2019-08
2017-03 2017-09 2018-03 2018-09 2019-03 2019-09
2017-04 2017-10 2018-04 2018-10 2019-04 2019-10
2017-05 2017-11 2018-05 2018-11 2019-05 2019-11
2017-06 2017-12 2018-06 2018-12 2019-06 2019-12
```

Pretty slick!

Parameter Expansion

We're only going to touch briefly on *parameter expansion* in this lesson, but we'll be covering it more 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 our examination. For example, the variable named “USER” contains our user name. To invoke parameter expansion and reveal the contents of USER we would do this:

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

To see a list of available variables, try this:

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

With other types of expansion, if we mistype a pattern, the expansion will not take place and the echo command will simply display the mistyped pattern. With parameter expansion, if we misspell the name of a variable, the expansion will still take place, but will result in an empty string:

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

Command Substitution

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

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

A clever one goes something like this:

```
[me@linuxbox me]$ 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 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 shown):

```
[me@linuxbox me]$ file $(ls /usr/bin/* | grep bin/zip)
/usr/bin/bunzip2:
/usr/bin/zip:      ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripp
/usr/bin/zipcloak: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripp
/usr/bin/zipgrep:  POSIX shell script text executable
/usr/bin/zipinfo:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripp
/usr/bin/zipnote:  ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripp
/usr/bin/zipsplit: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripp
```

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 which is also supported in **`bash`**. It

uses back-quotes instead of the dollar sign and parentheses:

```
[me@linuxbox me]$ 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:

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

or:

```
[me@linuxbox me]$ [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 “$”, “\” (backslash), and “`” (back- quote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Imagine 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 me]$ 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 can stop the word-splitting and get the desired result; further, we can even repair the damage:

```
[me@linuxbox me]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2020-02-20 13:03 two words.txt
[me@linuxbox me]$ 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 me]$ 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 me]$ echo this is a     test
this is a test
```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as delimiters between words. This means that unquoted spaces, tabs, and newlines are not considered to be part of the text. They only serve as separators. Since 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 me]$ echo "this is a     test"
this is a     test
```

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 me]$ echo $(cal)
February 2020 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox me]$  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 thirty-eight arguments. In the second, a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

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

```
[me@linuxbox me]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox me]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox me]$ 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 only want to quote 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 me]$ 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 “$”, “!”, “&”, “ “, and others. To include a special character in a filename we can to this:

```
[me@linuxbox me]$ 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.

More Backslash Tricks

If we look at the **man** pages for any program written by the GNU project, we will see that in addition to command line options consisting of a dash and a single letter, there are also long option names that begin with two dashes. For example, the following are equivalent:

```
ls -r
ls --reverse
```

Why do they support both? The short form is for lazy typists on the command line and the long form is mostly for scripts though some options may only be available in long form. Sometimes it is better to use a long option when the option is obscure or we want to document more clearly what an option is. This is especially useful when writing scripts where maximum readability is desired, and besides, anytime we can save ourselves a trip to the man page is a good thing.

As we might suspect, using the long form options can make a single command line very long. To combat this problem, we can use a backslash to get the shell to ignore a newline character like this:

```
ls -l \
   --reverse \
   --human-readable \
   --full-time
```

Using the backslash in this way allows us to embed newlines in our command. Note that for this trick to work, the newline must be typed immediately after the backslash. If we put a space after the backslash, the space will be ignored, not the newline. Backslashes are also used to insert special characters into our text. These are called *backslash escape characters*. Here are the common ones:

Escape Character	Name	Possible Uses
\n	newline	Adding blank lines to text
\t	tab	Inserting horizontal tabs to text
\a	alert	Makes our terminal beep
\\	backslash	Inserts a backslash
\f	formfeed	Sending this to our printer ejects the page

The use of the backslash escape characters is very common. This idea first appeared in the C programming language. Today, the shell, C++, Perl, python, awk, tcl, and many other programming languages use this concept. Using the **echo** command with the -e option will allow us to demonstrate:

```
[me@linuxbox me]$ echo -e "Inserting several blank lines\n\n\n"
Inserting several blank lines

[me@linuxbox me]$ echo -e "Words\tseparated\tby\thorizontal\ttabs."
Words separated   by  horizontal  tabs
[me@linuxbox me]$ echo -e "\aMy computer went \"beep\"."
My computer went "beep".
[me@linuxbox me]$ echo -e "DEL C:\\WIN2K\\LEGACY_OS.EXE"
DEL C:\WIN2K\LEGACY_OS.EXE
```