Stephen G. Kochan
Patrick Wood

Fourth Edition

Shell Programming in Unix, Linux and OS X

An iconic symbol of the American West, **Monument Valley** is one of the natural wonders of the world. The red-sand desert region is located within the range of the Navajo Nation on the Arizona-Utah border and is host to towering sandstone rock formations that have been sculpted over time and soar 400 to 1,000 feet above the valley floor. Three of the valley's most photographed peaks are the distinctive East and West Mitten Buttes and Merrick Butte.

5

Can I Quote You on That?

This chapter teaches you about a unique feature of the shell programming language: the way it interprets quote characters. The shell recognizes four different types of quote characters:

- The single quote character '
- The double quote character "
- The backslash character \
- The back quote character `

The first two and the last characters in the preceding list must occur in pairs, while the backslash character can be used any number of times in a command as needed. Each of these quotes has a distinct meaning to the shell. We'll cover them in separate sections of this chapter.

The Single Quote

There are many reasons that you'll need to use quotes in the shell. One of the most common is to keep character sequences that include whitespace together as a single element.

Here's a file called `phonebook` that contains names and phone numbers:

```
$ cat phonebook
Alice Chebba    973-555-2015
Barbara Swingle 201-555-9257
Liz Stachiw     212-555-2298
Susan Goldberg  201-555-7776
Susan Topple    212-555-4932
Tony Iannino    973-555-1295
$
```

To look up someone in our `phonebook` file you could use `grep`:

```
$ grep Alice phonebook
Alice Chebba    973-555-2015
$
```

Look what happens when you look up `Susan`:

```
$ grep Susan phonebook
Susan Goldberg  201-555-7776
Susan Topple    212-555-4932
$
```

There are two Susans in the datafile, hence the two lines of output—but suppose you only wanted Susan Goldberg's information. One way to overcome this problem would be to further qualify the name. For example, you could specify the last name as well:

```
$ grep Susan Goldberg phonebook
grep: can't open Goldberg
Susan Goldberg  201-555-7776
Susan Topple    212-555-4932
$
```

But that's not going to work, as you can see.

Why? Because the shell uses whitespace characters to separate the command arguments, the preceding command line results in `grep` being passed three arguments: `Susan`, `Goldberg`, and `phonebook` (see Figure 5.1).

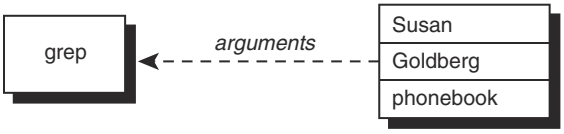

Figure 5.1 `grep Susan Goldberg phonebook`

When `grep` is executed, it interprets the first argument as the search pattern and the remaining arguments as the names of the files to search. In this case, `grep` thinks it's supposed to look for `Susan` in the files `Goldberg` and `phonebook`. It tries to open the file `Goldberg`, can't find it, and issues the error message:

```
grep: can't open Goldberg
```

Then it goes to the next file, `phonebook`, opens it, searches for the pattern `Susan`, and prints the two matching lines. Quite logical, really.

The problem is really about how to pass arguments that include whitespace characters to programs.

The solution: enclose the entire argument inside a pair of single quotes, as in

```
grep 'Susan Goldberg' phonebook
```

When the shell sees the first single quote, *it ignores any special characters that follow until it sees the matching closing quote*.

```
$ grep 'Susan Goldberg' phonebook
Susan Goldberg   201-555-7776
$
```

As soon as the shell encountered the first `'` it stopped interpreting any special characters until it found the closing `'`. So the space between `Susan` and `Goldberg`, which would have normally delimited two separate arguments, was ignored by the shell. The shell then split the command line into *two* arguments, the first `Susan Goldberg` (which includes the space character) and the second `phonebook`. It then invoked `grep`, passing it these two arguments (see Figure 5.2).

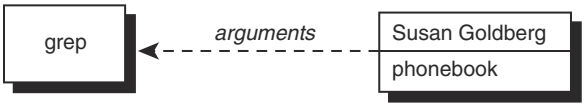

Figure 5.2 `grep 'Susan Goldberg' phonebook`

`grep` interpreted the first argument, `Susan Goldberg`, as a pattern that included an embedded space and looked for it in the file specified by the second argument, `phonebook`. Note that the shell *removes* the quotes and does not pass them to the program.

No matter how many space characters are enclosed between quotes, they are all preserved by the shell.

```
$ echo  one            two      three    four
one two three four
$ echo 'one            two      three    four'
one            two      three    four
$
```

In the first case, the shell removes the extra whitespace characters from the line (no quotes!) and passes `echo` the four arguments `one`, `two`, `three`, and `four` (see Figure 5.3).

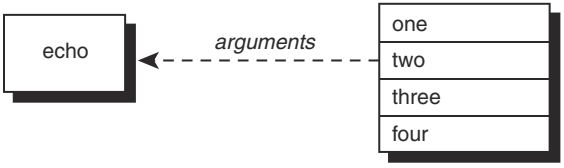

Figure 5.3 `echo  one  two  three  four`

In the second case, the extra spaces are preserved and the shell treats the entire string of characters enclosed in quotes as a single argument when executing `echo` (see Figure 5.4).

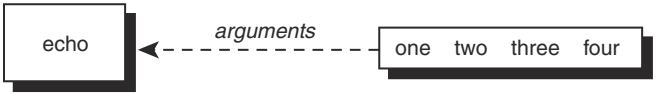

Figure 5.4 `echo  'one  two  three  four'`

Worth emphasizing is that *all* special characters are ignored by the shell if they appear within single quotes. That explains how the following works:

```
$ file=/users/steve/bin/progl
$ echo $file
/users/steve/bin/progl
$ echo '$file'                    $ not interpreted
$file
$ echo *
addresses intro lotsaspaces names nu numbers phonebook stat
$ echo '*'
*
$ echo '< > | ; ( ) { } >> " &'
< > | ; ( ) { } >> " &
$
```

Even the `Enter` key will be retained as part of the command argument if it's enclosed in single quotes:

```
$ echo 'How are you today,
> John'
How are you today,
John
$
```

After parsing the first line, the shell sees that the quote isn't matched, so it prompts the user (with `>`) to type in the closing quote. The `>` is known as the *secondary* prompt character and is displayed by the shell whenever it's waiting for you to finish typing a multi-line command.

Quotes are also needed when assigning values containing whitespace or special characters to shell variables, though there are nuances, as demonstrated:

```
$ message='I must say, this sure is fun'
$ echo $message
I must say, this sure is fun
$ text='* means all files in the directory'
$ echo $text
names nu numbers phonebook stat means all files in the directory
$
```

The quotes are needed in the first statement because the value being stored includes spaces.

The second sequence with the variable `text` highlights that the shell does filename substitution after variable name substitution, meaning that the `*` is replaced by the names of all the files in the current directory after the variable is expanded, but before the `echo` is executed. Annoying!

How do you fix these sort of problems? Through the use of double quotes.

The Double Quote

Double quotes work similarly to single quotes, except they're less protective of their content: single quotes tell the shell to ignore *all* enclosed characters, double quotes say to ignore *most*. In particular, the following three characters are not ignored inside double quotes:

- Dollar signs
- Back quotes
- Backslashes

The fact that dollar signs are not ignored means that variable name substitution is done by the shell inside double quotes.

```
$ filelist=*
$ echo $filelist
addresses intro lotsaspaces names nu numbers phonebook stat
$ echo '$filelist'
$filelist
$ echo "$filelist"
*
$
```

Here you see the major difference between no quotes, single quotes, and double quotes. In the first instance, the shell sees the asterisk and substitutes all the filenames from the current directory. In the second case, the shell leaves the characters enclosed within the single quotes completely alone, which results in the display of `$filelist`. In the final case, the double quotes indicate to the shell that variable name substitution is still to be performed inside the quotes. So the shell substitutes `*` for `$filelist`. But because filename substitution is *not* done inside double quotes, `*` is then safely passed to `echo` as the value to be displayed.

> **Note**
>
> While we're talking about single versus double quotes, you should also be aware that the shell has no idea what "smart quotes" are. Those are generated by word processors like Microsoft Word and curl "inward" towards the material they surround, making it much more attractive when printed. The problem is, that'll break your shell scripts, so be alert!

If you want to have the value of a variable substituted, but don't want the shell to then parse the substituted characters specially, enclose the variable inside double quotes.

Here's another example illustrating the difference between double quotes and no quotes:

```
$ address="39 East 12th Street
> New York, N. Y. 10003"
$ echo $address
39 East 12th Street New York, N. Y. 10003
$ echo "$address"
39 East 12th Street
New York, N. Y. 10003
$
```

Note that in this particular example, it makes no difference whether the value assigned to `address` is enclosed in single quotes or double quotes. The shell displays the secondary command prompt in either case to indicate it's waiting for the corresponding close quote.

After assigning the two-line address to `address`, the value of the variable is displayed by `echo`. Without the variable being quoted the address is displayed on a single line. The reason is the same as what caused

```
echo one           two       three    four
```

to be displayed as

```
one two three four
```

Because the shell removes spaces, tabs, and newlines (whitespace characters) from the command line and then cuts it up into arguments before giving it to the requested command, the invocation

```
echo $address
```

causes the shell to remove the embedded newline character, treating it as it would a space or tab: as an argument delimiter. Then the shell passes the *nine* arguments to `echo` for display. `echo` never sees that newline; the shell gets to it first (see Figure 5.5).

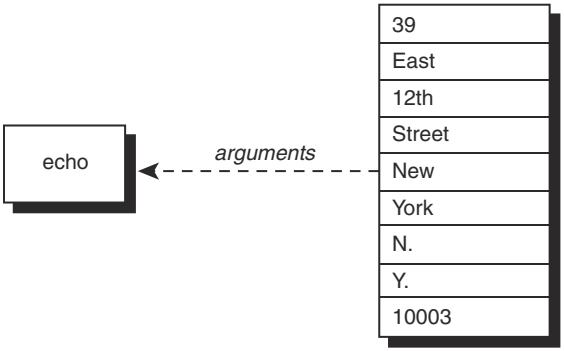

Figure 5.5 `echo $address`

When the command

```
echo "$address"
```

is used instead, the shell substitutes the value of `address` as before, except that the double quotes tell it to leave any embedded whitespace characters alone. So in this case, the shell passes a single argument to `echo`—an argument that contains an embedded newline. `echo` then displays its single argument. Figure 5.6 illustrates this, with the newline character depicted by the sequence `\n`.