

# Effects of Double Quotes

*in the*

## Unix Shell

*and*

## Perl

**Tim Maher, Consultix**

**tim@TeachMePerl.com    tim@TeachMeUnix.com**

The principal effect of double quotes in the Shell<sup>1</sup> and Perl is to process the indicated “substitutions” (or “interpolations”), if any, and then to form the quoted material into a single string. This article discusses the additional complications that apply in the Shell case, and shows Perl expressions that come as close as possible to duplicating the Shell’s behavior.

The main tool available to Shell programmers for controlling the word-splitting that otherwise occurs after other substitutions have been performed is the widely misunderstood “double quote”.<sup>2</sup> To help you understand it, Table 1 summarizes the effects of “no quoting” versus “double quoting” on substitutions in the Shell, and interpolations in Perl.

**The main tool available to Shell programmers for controlling the word-splitting that otherwise occurs after other substitutions is the widely misunderstood “double quote”.**

---

<sup>1</sup>When capitalized, Shell refers to the Bourne, Korn, Bash, and POSIX-compliant shell interpreters found on UNIX and UNIX-like (“Unix”) systems.

<sup>2</sup>Other tools that can be used to control automatic word-splitting include the IFS (Internal Field Separators) variable in the Shell, and the `$/` variable and `-a` and `-aF 'sep'` invocation options in Perl.

Table 1. Effects of double quotes on results of Shell substitutions and Perl interpolations

	Shell		Perl	
	No Quotes	Double Quotes	No Quotes	Double Quotes
<b>Filename Generation</b>	Works	Disallowed	Works	Disallowed
<b>Scalar Variables</b>	Results subjected to other substitutions and word-splitting. Result of NULL string is discarded.	Results treated as single "word". Result of NULL string is preserved.	Result treated as single word. NULL and undefined values preserved.	Same as No Quotes.
<b>Array Variables</b>	Results subjected to other substitutions and word-splitting. Result of NULL string is discarded.	Results treated as single "word", if * or numeric index is used, or one word per element, if @ index used. Result of NULL string is preserved.	In LIST context, result is a list; in scalar context, result is number of elements in list. NULL and undefined values preserved.	Value of \$ " variable (space character by default) is interpolated between elements. Result treated as scalar (single word). NULL and undefined values preserved.
<b>Command Substitution/ Interpolation</b>	Results subjected to other substitutions and word-splitting. Result of NULL string is discarded.	Results treated as single "word". Result of NULL string is preserved.	In LIST context, result is a list, with elements delimited by string in \$ / (" \n" by default). In scalar context, result is all output. NULL values preserved.	Disallowed

As indicated in the Shell/Scalar Variables/Double Quotes cell of the table, the Shell expression "\$names" yields all the names as a single string. However, as the Perl/ Scalar Variables/Double Quotes cell indicates, \$names yields a single string in Perl with or without double quotes, in keeping with Perl's default policy of not tampering with data without permission.

Therefore, in the usual case where you don't want word splitting on the results of your substitutions, you get that behavior automatically in Perl, and when you want word splitting, you only have to call the `split` function to get it.

Being an accurate reflection of the Shell's behavior, Table 1 is fairly complicated, so let us look at some of its implications. In the following examples, two-element arrays consisting of "Britney Spears" and "Cher" are used, along with a two-line file called `stars` that contains the same information, one singer per line.

**\$names yields a single string in Perl with or without double quotes, in keeping with Perl's default policy of not tampering with data without permission.**

**Shell:**

For instance, given an array of two elements, whose first element contains two words and the second a single word, the following `for`-loop headers yield different numbers of iterations according to the type of index that is used (`@` is “magical”), and the type of quoting employed:<sup>3</sup>

```
names=('Britney Spears' 'Cher')
for i in ${names[*]} ... # 3 iterations: Britney, Spears, Cher
for i in "${names[@]}" ... # 2 iterations; Britney Spears, Cher
```

The variable substitution request yielding the two-iteration case maintains the separation between the elements but disallows word-splitting within them, due to the special behavior of the `@` index within double quotes, whereas the unquoted case, which allows word-splitting, produces a separate iteration for each word. As you would imagine, the most commonly desired result is that of the second code sample, in which a separate iteration is provided for each array element.

We must not forget about command substitution requests, because they are another common list generator for these loops. Such requests are never quoted when used in a list-generation context, because that would always result in a single “word”. For this reason, a command substitution request would yield the same result as the unquoted variable case above:

```
for i in `cat stars` ... # 3 iterations: Britney, Spears, Cher
```

**Perl:**

Here are the Perl counterparts to the above looping arrangements. Note that the case that is by far the most commonly desired, the “2 iterations” one, is also the easiest to type—as it should be! However, if the splitting of lines into words is desired, that can be arranged through use of `split`:

```
@N=('Britney Spears', 'Cher');
foreach ( split /\s+/, "@N" ) ... # 3 iterations: B., S., Cher
foreach ( @N ) ... # 2 iterations: B. S., Cher
```

Does the use of double quoting on `@names` in connection with the `split` function seem odd to you? It very well may, because in the Shell quoting prevents word-splitting. The reason it is used in this Perl case is that `split` only works on scalars, so quoting is needed to join “Britney Spears” and “Cher” into a single string in preparation for extracting its words.

Here are the corresponding loop invocations for the special variables that contain the script’s arguments (again featuring “Britney Spears” and “Cher”):

**Shell:**

```
for i in $* ... # 3 iterations: B., S., Cher
for i in "$@" ... # 2 iterations: B. S., Cher
```

**Perl:**

```
foreach ( split /\s+/, "@ARGV" ) ... # 3 iterations: B., S., Cher
foreach ( @ARGV ) ... # 2 iterations: B. S., Cher
```

---

<sup>3</sup>To keep things as simple as possible, we will overlook the fact that other substitutions would be attempted on the results produced by the unquoted substitution. For example, filename substitution could change the results if a `*` appeared after Cher’s name, which could result in additional iterations, one per generated filename (Cheri, etc.). Cases that would result in a single iteration, such as the Shell’s `"${names[*]}"`, are also omitted from this discussion, because they would rarely be used (on purpose, anyway) with `for` or `foreach`.

To complete the set of examples, here are the corresponding loop invocations for user-defined scalar variables:

**Shell:**

```
names='Nigel Gareth Derek'
for i in      $names          ... # 3 iterations: N., G., Derek
```

**Perl:**

```
$names='Nigel Gareth Derek';
foreach ( split /\s+/, $names ) ... # 3 iterations: N., G., Derek
```

How do the languages compare? The Shell clearly provides the easier specification for obtaining 3 iterations, due to its eagerness to do word-splitting on words within scalar variables. This is an outgrowth of its historical lack of array variables, and its resulting need to treat scalars as containers for lists. However, from Perl's point of view, if you had wanted those names to be considered as list elements you should have stored them in an array in the first place, allowing the simple `foreach (@names)` syntax shown earlier, along with a simple way to treat the list as a scalar when necessary, using `foreach "@names"`.

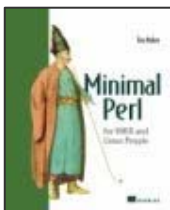
Some encouraging news is found in Table 1's upper right corner, which reports that double quotes have no affect whatsoever on scalar variables in Perl. This means your hard-earned habit of routinely double-quoting most Shell substitutions<sup>4</sup> will not interfere with your migration to Perl.

**Your hard-earned habit of routinely double-quoting most Shell substitutions will not interfere with your migration to Perl.**

The upshot of this discussion is simply that quoting is often needed in the Shell to disable unwanted processing of the results of substitutions, whereas in Perl additional processing is only applied if you ask for it.

Perl's approach is, in a word, *better*. In my experience, relatively few Shell programmers ever get to the point where they fully comprehend the Shell's processing model, and many experience rude awakenings when a long-dormant unquoted substitution suddenly starts generating unexpected results (for example, due to a file having arrived in the current directory that suddenly provides a filename-generation match for an asterisk, which results from an unquoted variable substitution).

In contrast, Perl programmers can sleep soundly knowing that their tried and tested interpolations won't suddenly mutate into alien monstrosities that will unexpectedly come back to haunt them at some future date.



For more of Dr. Maher's insights into the relationship between Perl and UNIX, see <http://MinimalPerl.com>. For worldwide hands-on training on Perl, see <http://TeachMePerl.com>.



<sup>4</sup>As recommended in <http://TeachMeUnix/quoting.html>.