Quotes and escaping

Quoting and escaping are important, as they influence the way Bash acts upon your input. There are three recognized types:

- **per-character escaping** using a backslash: `\$stuff`
- **weak quoting** with double-quotes: `"stuff"`
- **strong quoting** with single-quotes: `'stuff'`

All three forms have the very same purpose: **They give you general control over parsing, expansion and expansion results.**

Besides these basic variants, there are some special quoting methods (like interpreting ANSI-C escapes in a string) you'll meet below.

ATTENTION The quote characters (`"`, double quote and `'`, single quote) are a syntax element that influence parsing. It is not related to the quote characters passed as text to the command line! The syntax quotes are removed before the command is called! Example:

```
### NO NO NO: this passes three strings:
###      (1)  "my
###      (2)  multiword
###      (3)  argument"
MYARG="\"my multiword argument\""
somecommand $MYARG

### THIS IS NOT (!) THE SAME AS ###
command "my multiword argument"

### YOU NEED ###
MYARG="my multiword argument"
command "$MYARG"
```

Per-character escaping

Per-character escaping is useful in on expansions and substitutions. In general, a character that has a special meaning to Bash, like the dollar-sign (`$`) can be masked to not have a special meaning using the backslash:

```
echo \$HOME is set to \"$HOME\"
```

- `\$HOME` won't expand because it's not in variable-expansion syntax anymore
- The backslash changes the quotes into literals - otherwise Bash would interpret them

The sequence `\<newline>` (an unquoted backslash, followed by a `<newline>` character) is interpreted as **line continuation**. It is removed from the input stream and thus effectively ignored. Use it to beautify your code:

```
# escapestr_sed()
# read a stream from stdin and escape characters in text that could be interpreted as
# special characters by sed
escape_sed() {
 sed \
  -e 's/\//\\\//g' \
  -e 's/\&/\\\&/g'
}
```

The backslash can be used to mask every character that has a special meaning to bash. Exception: Inside a single-quoted string (see below).

Weak quoting

Inside a weak-quoted string there's **no special interpretion of**:

- spaces as word-separators (on inital commandline splitting and on word splitting!)
- single-quotes to introduce strong-quoting (see below)
- characters for pattern matching
- pathname expansion
- process substitution

Everything else, especially parameter expansion, is performed!

```
ls -l "*"
```

Will not be expanded. `ls` gets the literal `*` as argument. It will, unless you have a file named `*`, spit out an error.

```
echo "Your PATH is: $PATH"
```

Will work as expected. `$PATH` is expanded, because it's double (weak) quoted.
If a backslash in double quotes ("weak quoting") occurs, there are 2 ways to deal with it

- if the baskslash is followed by a character that would have a special meaning even inside double-quotes, the backslash is removed and the following character looses its special meaning
- if the backslash is followed by a character without special meaning, the backslash is not removed

In particuar this means that `"\$"` will become `$`, but `"\x"` will become `\x`.

Strong quoting

Strong quoting is very easy to explain:

Inside a single-quoted string **nothing** is interpreted, except the single-quote that closes the string.

```
echo 'Your PATH is: $PATH'
```

`$PATH` won't be expanded, it's interpreted as ordinary text because it's surrounded by strong quotes.
In practise that means, to produce a text like `Here's my test…` as a single-quoted string, you have to leave and re-enter the single quoting to get the character "`'`" as literal text:

```
# WRONG
echo 'Here's my test...'


# RIGHT
echo 'Here'\''s my test...'


# ALTERNATIVE: It's also possible to mix-and-match quotes for readability:
echo "Here's my test"
```

ANSI C like strings

Bash provides another quoting mechanism: Strings that contain ANSI C-like escape sequences. The Syntax is:

```
$'string'
```

where the following escape sequences are decoded in `string`:

Code	Meaning
`\"`	double-quote
`\'`	single-quote
`\\`	backslash
`\a`	terminal alert character (bell)
`\b`	backspace
`\e`	escape (ASCII 033)
`\E`	escape (ASCII 033) **\E is non-standard**
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\cx`	a control-x character, for example, `$'\cZ'` to print the control sequence composed of Ctrl-Z (`^Z`)
`\uXXXX`	Interprets `xxxx` as a hexadecimal number and prints the corresponding character from the character s alpha)
`\UXXXXXXXX`	Interprets `xxxx` as a hexadecimal number and prints the corresponding character from the character s alpha)

Code	Meaning
`\nnn`	the eight-bit character whose value is the octal value nnn (one to three digits)
`\xHH`	the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

This is especially useful when you want to pass special characters as arguments to some programs, like passing a newline to sed.

The resulting text is treated as if it were **single-quoted**. No further expansion happens.

The `$'...'` syntax comes from ksh93, but is portable to most modern shells including pdksh. A specification for it was accepted for SUS issue 7. There are still some stragglers, such as most ash variants including dash, (except busybox built with "bash compatibility" features).

I18N/L10N

A dollar-sign followed by a double-quoted string, for example

```
echo $"generating database..."
```

means I18N. If there is a translation available for that string, it is used instead of the given text. If not, or if the locale is `C`/`POSIX`, the dollar sign is simply ignored, which results in a normal double quoted string.
If the string was replaced (translated), the result is double quoted.

In case you're a C programmer: The purpose of `$"…"` is the same as for `gettext()` or `_()`.

For useful examples to localize your scripts, please see Appendix I of the Advanced Bash Scripting Guide.

Attention: There is a security hole. Please read the gettext documentation

Common mistakes

String lists in for-loops

The classic for loop uses a list of words to iterate through. The list can also be in a variable:

```
mylist="DOG CAT BIRD HORSE"
```

WRONG way to iterate through this list:

```
for animal in "$mylist"; do
    echo $animal
done
```

Why? Due to the double-quotes, technically, the expansion of `$mylist` is seen as **one word**. The for loop iterates exactly one time, with `animal` set to the whole list.
RIGHT way to iterate through this list:

```
for animal in $mylist; do
    echo $animal
done
```

Working out the test-command

The command `test` or `[ … ]` (the classic test command) is an ordinary command, so ordinary syntax rules apply. Let's take string comparison as an example:

```
[ WORD = WORD ]
```

The `]` at the end is a convenience; if you type `which [` you will see that there is in fact a binary file with that name. So if we were writing this as a test command it would be:

```
test WORD = WORD
```

When you compare variables, it's wise to quote them. Let's create a test string with spaces:

```
mystring="my string"
```

And now check that string against the word "testword":

```
[ $mystring = testword ] # WRONG!
```

This fails! These are too many arguments for the string comparison test. After expansion is performed, you really execute:

```
[ my string = testword ]
test my string = testword
```

Which is wrong, because `my` and `string` are two separate arguments.
So what you really want to do is:

```
[ "$mystring" = testword ] # RIGHT!
```

```
test 'my string' = testword
```

Now the command has three parameters, which makes sense for a binary (two argument) operator.

Hint: Inside the conditional expression (`[[ ]]`) Bash doesn't perform word splitting, and thus you don't need to quote your variable references - they are always seen as "one word".