Command substitution

```
$( <COMMANDS> )

` <COMMANDS> `
```

The command substitution expands to the output of commands. These commands are executed in a subshell, and their `stdout` data is what the substitution syntax expands to.

All **trailing** newlines are removed (below is an example for a workaround).

In later steps, **if not quoted**, the results undergo word splitting and pathname expansion. You have to remember that, because the word splitting will also remove embedded newlines and other `IFS` characters and break the results up into several words. Also you'll probably get unexpected pathname matches. **If you need the literal results, quote the command substitution!**

The second form `` `COMMAND` `` is more or less obsolete for Bash, since it has some trouble with nesting ("inner" backticks need to be escaped) and escaping characters. Use `$(COMMAND)` , it's also POSIX!

When you call an explicit subshell `(COMMAND)` inside the command substitution `$()` , then take care, this way is **wrong**:

```
$((COMMAND))
```

Why? because it collides with the syntax for arithmetic expansion. You need to separate the command substitution from the inner `(COMMAND)` :

```
$( (COMMAND) )
```

Specialities

When the inner command is only an input redirection, and nothing else, for example

```
$( <FILE )
# or
` <FILE `
```

then Bash attempts to read the given file and act just if the given command was `cat FILE` .

A closer look at the two forms

In general you really should only use the form `$()`, it's escaping-neutral, it's nestable, it's also POSIX. But take a look at the following code snips to decide yourself which form you need under specific circumstances:

Nesting

Backtick form `` `…` `` is not directly nestable. You will have to escape the "inner" backticks. Also, the deeper you go, the more escape characters you need. Ugly.

```
echo `echo `ls``      # INCORRECT
echo `echo \`ls\``    # CORRECT
echo $(echo $(ls))    # CORRECT
```

Parsing

All is based on the fact that the backquote-form is simple character substitution, while every `$()` -construct opens an own, subsequent parsing step. Everything inside `$()` is interpreted as if written normal on a commandline. No special escaping of **nothing** is needed:

```
echo "$(echo "$(ls)")" # nested double-quotes - no problem
```

Constructs you should avoid

It's not all shiny with `$()`, at least for my current Bash (`3.1.17(1)-release`. **Update:** Fixed since `3.2-beta` together with a misinterpretion of '))' being recognized as arithmetic expansion [by redduck666]). This command seems to incorrectly close the substitution step and echo prints "ls" and ")":

```
echo $(
# some comment ending with a )
ls
)
```

It seems that every closing ")" confuses this construct. Also a (very uncommon) construct like:

```
echo $(read VAR; case "$var" in foo) blah ;; esac) # spits out some e
rror, when it sees the ";;"

# fixes it:
echo $(read VAR; case "$var" in (foo) blah ;; esac) # will work, but
just let it be, please ;-)
```

Conclusion:

In general, the `$()` should be the preferred method:

- it's clean syntax
- it's intuitive syntax
- it's more readable
- it's nestable
- its inner parsing is separate

Examples

To get the date:

```
DATE="$(date)"
```

To copy a file and get `cp` error output:

```
COPY_OUTPUT="$(cp file.txt /some/where 2>&1)"
```

Attention: Here, you need to redirect `cp` `STDERR` to its `STDOUT` target, because command substitution only catches `STDOUT`!

Catch stdout and preserve trailing newlines:

```
var=$(echo -n $'\n'); echo -n "$var"; # $var == ""
var=$(echo -n $'\n'; echo -n x); var="${var%x}"; echo -n "$var" # $va
r == "\n"
```

This adds "x" to the output, which prevents the trailing newlines of the previous commands' output from being deleted by $().

By removing this "x" later on, we are left with the previous commands' output with its trailing newlines.

See also

- Internal: Introduction to expansion and substitution
- Internal: Obsolete and deprecated syntax

Discussion

paul, 2010/06/01 11:51 (), 2010/06/02 04:21 ()

Great article, I wonder if you can help with something...

I am doing a pentest and have found a dir traversal (and so have retrevied the source for the faulty script) but whatever I try, I can't break out of the command substitution and get a command injection... It's _got_ to be possible....

Here are the faulty lines fro the script;

```
#!/bin/sh
/bin/echo "<h2>Audit log for " $QUERY_STRING "</h2>"
/bin/cat /mnt/video0/LOGS/`echo $QUERY_STRING | sed "s/-//g"`.ext
```

...the dir traversal is done by just sending the file to read followed by a space and any character (so the .ext appends and creates two calls to the cat command, the second of which fails).

...any ideas ?

Thanks, Pabb

Tony (http://wiki.bash-hackers.org/syntax/expansion/cmdsubst), 2010/09/09 16:10 ()

Hello

Usually I see the trailing quotation mark matching the leading quotation mark e.g. ‘ followed by ‘. This is also true with the backtick.

This stayed true until I journeyed through some depreciated UNIX literature where the backtick was followed by the single quote.

Now if I have read your <specialities section> correctly this deviates when the inner command is only used for input redirection. Is this a syntax shorthand or are there any side effects to using the original balanced pair mode of backtick quotation marks which I have failed to pick up?

e.g. ` <FILE ` VERSUS ` <FILE '

Jan Schampera, 2010/09/09 16:49 ()

I'm so sorry, this was a typo.

The notation you found in the old literature is a very common text-only notation (nothing technical). Maybe they like to use it because while reading it can't be mixed with (technical) shell quotes.

For example see the bash(1) manpage:

```
Brace  expansions  may  be  nested.  The results of each expan
ded string are not sorted; left to right order is preserved.
For example, a{d,c,b}e expands into `ade ace abe'.
```

Gregg, 2011/12/31 09:01 ()

Why doesn't this work?

```
echo $(sed 's/local/$HOME\//' /etc/hosts)
```

I would expect 'localhost' to become '/home/user/host/' but it becomes '$HOME/host'

Thanks

Jan Schampera, 2011/12/31 16:51 ()

Hi,

because of the quoting. Use normal double quotes there, to make Bash replacing `$HOME` , also use another delimiter for sed (here it's the hashmark):

```
sed "s#local#$HOME/#" /etc/hosts
```

Nick Nax, 2012/03/18 06:04 (), 2012/03/18 14:03 ()

You are correct that ")" confuses the $() command substitution syntax.

For example, this works

```
s=`for((i=0;i<3;i++)); do case $x in a) echo -n A ;; *) echo -n B
;; esac; done
```

while this does NOT work:

```
s=$( for((i=0;i<3;i++)); do case $x in a) echo -n A;; *) echo -n
B;; esac; done)
```

syntax/expansion/cmdsubst.txt Last modified: 2010/09/09 16:45 (external edit)

Bash Hackers Wiki