How to use double or single brackets, parentheses, curly braces

6 Answers

In Bash, `test` and `[` are builtins.

The double bracket enables additional functionality. For example, you can use `&&` and `||` instead of `-a` and `-o` and there's a regular expression matching operator `=~`.

The braces, in addition to delimiting a variable name are used for parameter expansion so you can do things like:

- Truncate the contents of a variable

```
$ var="abcde"; echo
${var%d*}
abc
```

- Make substitutions similar to `sed`

```
$ var="abcde"; echo
${var/de/12}
abc12
```

- Use a default value

```
$ default="hello"; unset var; echo ${var:-
$default}
hello
```

- and several more

Also, brace expansions create lists of strings which are typically iterated over in loops:

```
$ echo f{oo,ee,a}d
food feed fad

$ mv error.log{,.OLD}
(error.log is renamed to error.log.OLD because the brace
expression
expands to "mv error.log error.log.OLD")

$ for num in {000..2}; do echo "$num"; done
000
001
002

$ echo {00..8..2}
00 02 04 06 08

$ echo {D..T..4}
D H L P T
```

Note that the leading zero and increment features weren't available before Bash 4.

Thanks to gboffi for reminding me about brace expansions.

Double parentheses are used for arithmetic operations:

```
((a++))

((meaning = 42))

for ((i=0; i<10; i++))

echo $((a + b + (14 *
c)))
```

and they enable you to omit the dollar signs on integer and array variables and include spaces around operators for readability.

Single brackets are also used for array indices:

```
array[4]="hello"

element=${array[index]}
```

Curly brace are required for (most/all?) array references on the right hand side.

ephemient's comment reminded me that parentheses are also used for subshells. And that they are used to create arrays.

```
array=(1 2 3)
echo
${array[1]}
2
```

1. A single bracket (`[`) usually actually calls a program named `[`; `man test` or `man [` for more info. Example:

```
$ VARIABLE=abcdef
$ if [ $VARIABLE == abcdef ] ; then echo yes ; else echo no ;
fi
yes
```

2. The double bracket (`[[`) does the same thing (basically) as a single bracket, but is a bash builtin.

```
$ VARIABLE=abcdef
$ if [[ $VARIABLE == 123456 ]] ; then echo yes ; else echo no ;
fi
no
```

3. Parentheses (`()`) are used to create a subshell. For example:

```
$ pwd
/home/user
$ (cd /tmp;
pwd)
/tmp
$ pwd
/home/user
```

   As you can see, the subshell allowed you to perform operations without affecting the environment of the current shell.

4a. Braces (`{}`) are used to unambiguously identify variables. Example:

```
    $ VARIABLE=abcdef
    $ echo Variable: $VARIABLE
    Variable: abcdef
    $ echo Variable: $VARIABLE123456
    Variable:
    $ echo Variable:
${VARIABLE}123456
    Variable: abcdef123456
```

4b. Braces are also used to execute a sequence of commands in the *current* shell context, e.g.

```
    $ { date; top -b -n1 | head ; } >logfile
    # 'date' and 'top' output are concatenated,
    # could be useful sometimes to hunt for a top loader )

    $ { date; make 2>&1; date; } | tee logfile
    # now we can calculate the duration of a build from the
logfile
```

There is a subtle syntactic difference with `(` `)`, though (see bash reference) ; essentially, a semicolon `;` after the last command within braces is a must, and the braces `{`, `}` **must** be surrounded by spaces.

Brackets

```
if [ CONDITION ]    Test construct
if [[ CONDITION ]]  Extended test construct
Array[1]=element1   Array initialization
[a-z]               Range of characters within a Regular Expression
$[ expression ]     A non-standard & obsolete version of $(( expression ))
[1]
```

[1] http://wiki.bash-hackers.org/scripting/obsolete

Curly Braces

```
${variable}                              Parameter substitution
${!variable}                             Indirect variable reference
{ command1; command2; . . . commandN; } Block of code
{string1,string2,string3,...}            Brace expansion
{a..z}                                   Extended brace expansion
{}                                       Text replacement, after find and
xargs
```

Parentheses

```
( command1; command2 )             Command group executed within a subshell
Array=(element1 element2 element3) Array initialization
result=$(COMMAND)                  Command substitution, new style
>(COMMAND)                         Process substitution
<(COMMAND)                         Process substitution
```

Double Parentheses

```
(( var = 78 ))            Integer arithmetic
var=$(( 20 + 5 ))         Integer arithmetic, with variable assignment
(( var++ ))               C-style variable increment
(( var-- ))               C-style variable decrement
(( var0 = var1<98?9:21 )) C-style ternary operation
```

I just wanted to add these from TLDP:

```
~:$ echo $SHELL
/bin/bash

~:$ echo ${#SHELL}
9

~:$ ARRAY=(one two three)

~:$ echo ${#ARRAY}
3
```

```
~:$ echo ${TEST:-test}
test

~:$ echo $TEST


~:$ export TEST=a_string

~:$ echo ${TEST:-test}
a_string

~:$ echo ${TEST2:-$TEST}
a_string

~:$ echo $TEST2


~:$ echo ${TEST2:=$TEST}
a_string

~:$ echo $TEST2
a_string

~:$ export
STRING="thisisaverylongname"

~:$ echo ${STRING:4}
isaverylongname

~:$ echo ${STRING:6:5}
avery

~:$ echo ${ARRAY[*]}
one two one three one four

~:$ echo ${ARRAY[*]#one}
two three four

~:$ echo ${ARRAY[*]#t}
one wo one hree one four

~:$ echo ${ARRAY[*]#t*}
one wo one hree one four

~:$ echo ${ARRAY[*]##t*}
one one one four

~:$ echo $STRING
thisisaverylongname

~:$ echo ${STRING%name}
thisisaverylong

~:$ echo ${STRING/name/string}
thisisaverylongstring
```

The difference between **test**, **[** and **[[** is explained in great details in the BashFAQ.

> *To cut a long story short: test implements the old, portable syntax of the command. In almost all shells (the oldest Bourne shells are the exception), [is a synonym for test (but requires a final argument of]). Although all modern shells have built-in implementations of [, there usually still is an external executable of that name, e.g. /bin/[.*
>
> *[[is a new improved version of it, which is a keyword, not a program. This has beneficial effects on the ease of use, as shown below. [[is understood by KornShell and BASH (e.g. 2.03), but not by the older POSIX or BourneShell.*

And the conclusion:

> *When should the new test command [[be used, and when the old one [? If portability to the BourneShell is a concern, the old syntax should be used. If on the other hand the script requires BASH or KornShell, the new syntax is much more flexible.*

Parentheses in function definition

Parentheses `()` are being used in function definition:

```
function_name () { command1 ; command2 ;
}
```

That is the reason you have to escape parentheses even in command parameters:

```
$ echo (
bash: syntax error near unexpected token `newline'

$ echo \(
(

$ echo () { command echo The command echo was redefined. ;
}
$ echo anything
The command echo was redefined.
```