

Conditional Operators

- test utility
- [], [[]] and (())



nditional operators

Conditional operators (conditionals) are the essential part of almost every existing programming language, Bash is no exception.

We use conditionals in conditional statement (if) when we want to implement logical branching in code, make it execute different code blocks judging by, e.g., the value of a certain variable, or it's existence in the first place.

There're also things called loops, that are used when you need to repeat some piece of code several times based on some condition. Conditional operators are also used in loops.

Conditional statement:

- if

Loops:

- for
- while
- until



block – A built-in command (the opening bracket is the command's name). Lacks functionality but is portable to any other shell. This is an alias to the **test** command, sort of a syntax sugar.

[[]] block - A **bash** syntax construct (a keyword). Will not work in older Bourne-compatible shells (sh, ksh, etc.), but has more powers, like regexp. comparison, wildcards, etc. Also it is considered safer to use.

Both of them return some numerical value, as Bash don't have Boolean type variables (True/False) Instead it checks a thing called exit codes. Summing all this up,

[] and [[]] operators both return exit code 0, if the statement inside is true, and 1 if false.

```
bash-3.2$ [[ foo = bar ]]
bash-3.2$ echo $?
1
bash-3.2$ [[ foo = foo ]]
bash-3.2$ echo $?
0
```

General rule: Use [] if you don't need complex comparison, else use [[]].

est operator usage

test command has the same capabilities as [, as the bracket is the alias for the test command. However, the usage differs, as you don't have to finish the test command entry with closing bracket.

Form man:

test expression

[expression]

DESCRIPTION

The **test** utility evaluates the <u>expression</u> and, if it evaluates to true, returns a zero (true) exit status; otherwise, it returns 1 (false). If there is no expression, test also returns 1 (false).

As you can see, you can use **test** expression almost the same way you use [

String comparison operators

Let's assume that **FOO="pizza",** then:

Operator	Description	Example
== or =	comparing strings on equality	[[\$FOO == "pizza"]] (true)
!=	comparing strings on inequality	[[\$FOO != "bar"]] (true)
<	lexicographic comparison (before)	[[\$FOO < "eggs"]] (false)
>	lexicographic comparison (after)	[[\$FOO > "ham"]] (true)
=~	Regexp match (Bash 3.x only)	[[\$FOO =~ .*zz.*]] (true)
-Z	testing string for having zero length	[[-z \$FOO]] (false)
-n	testing string for having non-zero length	[[-n \$FOO]] (true)

eger comparison operators

Let's assume that **FOO=2020**, then:

Operator	Description	Example
-eq	comparing integers on equality	[[\$FOO –eq 3]] (false)
-ne	comparing integers on inequality	[[\$FOO -ne 3]] (true)
-lt	integer less than	[[\$FOO –lt 2020]] (false)
-le	integer less or equal to	[[\$FOO -le 2020]] (true)
-gt	integer greater than	[[\$FOO –gt 2020]] (false)
-ge	integer greater or equal to	[[\$FOO –ge 2020]] (true)

nditional operator usage

```
bash-3.2$ cat ./example_1.sh
#!/bin/bash
if test -z $1 || [ -z $2 ]; then
  echo "This script requires 2 arguments, got $#"
else
  echo "At least 2 variables are set"
fi
if [[ $1 > $2 ]]; then
  echo "1st argument is bigger than 2nd"
elif \lceil \lceil \$2 > \$1 \rceil \rceil; then
  echo "2nd argument is bigger than 1st"
else
  echo "arguments are equal"
```

bash-3.2\$./example_1.sh This script requires 2 arguments, got 0 arguments are equal bash-3.2\$./example_1.sh 10 This script requires 2 arguments, got 1 1st argument is bigger than 2nd bash-3.2\$./example_1.sh 10 20 At least 2 variables are set 2nd argument is bigger than 1st bash-3.2\$./example_1.sh 20 10 At least 2 variables are set 1st argument is bigger than 2nd

Example script

Result



```
bash-3.2$ if echo foo | grep bar; then echo "Conditional resulted in 0"; fi bash-3.2$ if echo foo | grep foo; then echo "Conditional resulted in 0"; fi foo Conditional resulted in 0
```



Arithmetic operator

(()) is used to do arithmetic calculations. This is used if For cycle in form of:

It has a different approach to exit codes:

If the expression inside (()) results in 0, the exit code will be 1 (False) If the expression results in anything else, the exit code will be 0 (True)

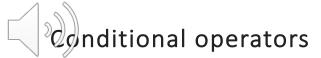
de with arithmetic operator example

```
bash-3.2$ cat ./example_2.sh
#!/bin/bash
if (( $1 < 10 )); then
  echo "argument is less than 10"
else
  echo "argument is more or equal to 10"
fi
(( 10 - $1 )); echo "Exit code of (( 10 - $1 )) is $?"
for ((i=\$1; i>0; i--)); do
  echo -n "$i "
done
echo
echo "Weird math resulted in $(( $1 + ($1*20) - ($1^2) ))"
```

```
bash-3.2$ ./example_2.sh 10
argument is more or equal to 10
Exit code of (( 10 - 10 )) is 1
10 9 8 7 6 5 4 3 2 1
Weird math resulted in 202
bash-3.2$ ./example_2.sh 15
argument is more or equal to 10
Exit code of (( 10 - 15 )) is 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Weird math resulted in 302
bash-3.2$ ./example_2.sh -10
argument is less than 10
Exit code of (( 10 - -10 )) is 0
Weird math resulted in -198
```

Example script

Result



Thanks for watching!

