# Brackets

Brackets

if [ CONDITION ] Test construct

if [[ CONDITION ]] Bash 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]

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

→ result=`COMMAND` (old style)

>(COMMAND) Process substitution

<(COMMAND) Process substitution

Double Parentheses

(( var = 78 )) Integer arithmetic

var=$(( 20 + 5 )) Integer arithmetic, with variable assignment

var=`expr 20 + 5`

var=$(expr 20 + 5)

(( var++ )) C-style variable increment

(( var-- )) C-style variable decrement

(( var0 = var1<98?9:21 )) C-style ternary operation

# [ ] vs. [[ ]]

POSIX vs Bash extension:

* + [ [is POSIX](http://pubs.opengroup.org/onlinepubs/9699919799/utilities/test.html)
  + [[ [is a Bash extension](https://www.gnu.org/software/bash/manual/html_node/Conditional-Constructs.html#index-_005b_005b) inspired from [KornShell](https://en.wikipedia.org/wiki/KornShell)
* regular command vs magic
  + [ is just a regular command with a weird name.

] is just the last argument of [.

[Ubuntu 16.04](https://en.wikipedia.org/wiki/Ubuntu_version_history#Ubuntu_16.04_LTS_.28Xenial_Xerus.29) actually has an executable for it at /usr/bin/[ provided by [coreutils](https://www.gnu.org/software/coreutils/manual/html_node/test-invocation.html#test-invocation), but the Bash built-in version takes precedence.

Nothing is altered in the way that Bash parses the command.

In particular, < is redirection, && and || concatenate multiple commands, ( ) generates subshells unless escaped by \, and word expansion happens as usual.

* + [[ X ]] is a single construct that makes X be parsed magically. <, &&, || and () are treated specially, and word splitting rules are different.

There are also further differences like = and =~.

In Bashese: [ is a built-in command, and [[ is a keyword: [*What's the difference between shell builtin and shell keyword?*](https://askubuntu.com/questions/445749/whats-the-difference-between-shell-builtin-and-shell-keyword)

* <
  + [[ a < b ]]: lexicographical comparison
  + [ a \< b ]: Same as above. \ required or else does redirection like for any other command. Bash extension.
  + expr x"$x" \< x"$y" > /dev/null or [ "$(expr x"$x" \< x"$y")" = 1 ]: POSIX equivalents, see: [*How to test strings for lexicographic less than or equal in Bash?*](https://stackoverflow.com/questions/21294867/how-to-test-strings-for-lexicographic-less-than-or-equal-in-bash/52707989#52707989)
* && and ||
  + [[ a = a && b = b ]]: true, logical and
  + [ a = a && b = b ]: syntax error, && parsed as an AND command separator cmd1 && cmd2
  + [ a = a ] && [ b = b ]: POSIX reliable equivalent
  + [ a = a -a b = b ]: almost equivalent, but deprecated by POSIX because it is insane and fails for some values of a or b like ! or ( which would be interpreted as logical operations
* (
  + [[ (a = a || a = b) && a = b ]]: false. Without ( ) it would be true, because [[ && ]] has greater precedence than [[ || ]]
  + [ ( a = a ) ]: syntax error, () is interpreted as a subshell
  + [ \( a = a -o a = b \) -a a = b ]: equivalent, but (), -a, and -o are deprecated by POSIX. Without \( \) it would be true, because -a has greater precedence than -o
  + { [ a = a ] || [ a = b ]; } && [ a = b ] non-deprecated POSIX equivalent. In this particular case however, we could have written just: [ a = a ] || [ a = b ] && [ a = b ], because the || and && shell operators have equal precedence, unlike [[ || ]] and [[ && ]] and -o, -a and [
* word splitting and filename generation upon expansions (split+glob)
  + x='a b'; [[ $x = 'a b' ]]: true. Quotes are not needed
  + x='a b'; [ $x = 'a b' ]: syntax error. It expands to [ a b = 'a b' ]
  + x='\*'; [ $x = 'a b' ]: syntax error if there's more than one file in the current directory.
  + x='a b'; [ "$x" = 'a b' ]: POSIX equivalent
* =
  + [[ ab = a? ]]: true, because it does [pattern matching](http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_13) (\* ? [ are magic). Does not glob expand to files in the current directory.
  + [ ab = a? ]: a? glob expands. So it may be true or false depending on the files in the current directory.
  + [ ab = a\? ]: false, not glob expansion
  + = and == are the same in both [ and [[, but == is a Bash extension.
  + case ab in (a?) echo match; esac: POSIX equivalent
  + [[ ab =~ 'ab?' ]]: false, loses magic with '' in Bash 3.2 and above and provided compatibility to Bash 3.1 is not enabled (like with BASH\_COMPAT=3.1)
  + [[ ab? =~ 'ab?' ]]: true
* =~
  + [[ ab =~ ab? ]]: true. POSIX [extended regular expression](http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html#tag_09_04) match and ? does not glob expand
  + [ a =~ a ]: syntax error. No Bash equivalent.
  + printf 'ab\n' | grep -Eq 'ab?': POSIX equivalent (single-line data only)

# `command` vs. $(command)

Command substitution allows the output of a command to replace the command name. There are two forms:

$(command)

or

`command`

Also,`` are more difficult to handle, you cannot nest them for example

# double vs. single quotes

$, ‘, “, \ are special characters in double quotes. Need to be escaped if to be treated as normal string.

|  |  |  |  |
| --- | --- | --- | --- |
| " | The double quote | The double quote ( "quote" ) protects everything enclosed between two double quote marks except $, ', " and \.Use the double quotes when you want only **variables and command substitution**. \* **Variable** - Yes \* **Wildcards** - No \* **Command substitution** - yes | The double quotes allowes to print the value of $SHELL variable, disables the meaning of [wildcards](https://bash.cyberciti.biz/guide/Wildcards), and finally allows command substitution. echo "$SHELL" echo "/etc/\*.conf" echo "Today is $(date)" |
| ' | The single quote | The single quote ( 'quote' ) protects everything enclosed between two single quote marks. It is used to **turn off the special meaning** of all characters. \* **Variable** - No \* **Wildcards** - No \* **Command substitution** - No | The single quotes prevents displaying variable $SHELL value, disabled the meaning of [wildcards](https://bash.cyberciti.biz/guide/Wildcards) /etc/\*.conf, and finally command substitution ($date) itself. echo '$SHELL' echo '/etc/\*.conf' echo 'Today is $(date)' |
| \ | The Backslash | Use backslash to change the special meaning of the characters or to escape special characters within the text such as quotation marks. | You can use \ before dollar sign to tell the shell to have no special meaning. Disable the meaning of the next character in $PATH (i.e. do not display value of $PATH variable): echo "Path is \$PATH" echo "Path is $PATH" |

# redirect to a file

echo “hello” > text.txt

> the output redirection operator used for overwriting files that already exist in the directory.

>> an output operator as well, but, it appends the data of an existing file

# input (<<< vs. << vs. <)

## <<< denotes a here string.

$ cat <<< 'hi there'

Output:

hi there

It passes the word on the right to the standard input of the command on the left.

## << denotes a here document.

$ cat <<EOF

> hi

> there

> EOF

hi

there

EOF can be any word.

Here documents are commonly used in shell scripts to create whole files or to display long messages.

cat > some-file <<FILE

foo

bar

bar bar

foo foo

FILE

## < FILE

passes the contents of a file to a command's standard input.

$ cat < /etc/fstab

/dev/sda2 /boot ext4 nosuid,noexec,nodev,rw,noatime,nodiratime 0 2

/dev/sda4 / ext4 rw,noatime,nodiratime, 0 1

/dev/sdb5 /var ext4 nosuid,noexec,nodev,rw,relatime 0 2

...

# list valid login shells

cat /etc/shells

# shebang:

#! /bin/bash

# comment

# this is a comment line

:’

line1

line2

‘

# variable

## env variable

export <NAME>=<VALUE>

## check env variable

echo $<NAME>

## evaluate an expression

`expr`

equivalent to

$(expr)

`1+1` → $(1+1)  
 ## define a variable  
 VAR1="Zara Ali"

VAR2=100

## read-only variables

NAME="Zara Ali"

readonly NAME

NAME="Qadiri" # error

## unset a variable

NAME=”Zara Ali”

unset NAME

## refer a variable

$VAR1

# Special variables:

|  |  |
| --- | --- |
| 1 | **$0**  The filename of the current script. |
| 2 | **$n**  These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). |
| 3 | **$#**  The number of arguments supplied to a script. |
| 4 | **$\***  All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. |
| 5 | **$@**  All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2. |
| 6 | **$?**  The exit status of the last command executed. |
| 7 | **$$**  The process number of the current shell. For shell scripts, this is the process ID under which they are executing. |
| 8 | **$!**  The process number of the last background command. |

## note:

## Special Parameters $\* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$\*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$\*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

# Array

## define:

array\_name[index]=value

array\_name=(value1 value2 value3)

## access an element:

${array\_name[index]}

## access all elements

You can access all the items in an array in one of the following ways –

(like Javascript spread operator)

${array\_name[\*]} → 1 2 3

${array\_name[@]}

## give back indices:

${!array\_name[@]}

## give back array length:

${#array\_name[@]}

## delete an element

unset array\_name[1]

## reset an element

array\_name[1]=another\_thing

# Basic operators

https://www.tutorialspoint.com/unix/unix-basic-operators.htm

# Conditional statement

if [ expression1 ]

then

statement1

statement2

.

.

elif [ expression2 ]

then

statement3

statement4

.

.

else

statement5

fi

CARS="bmw"

#Pass the variable in string

case "$CARS" in

# case 1

"mercedes") echo "Headquarters - Affalterbach, Germany" ;;

# case 2

"audi") echo "Headquarters - Ingolstadt, Germany" ;;

# case 3

"bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;

esac

# Loop

## while

while [ expression1 ]

do

statement1

done

## until

until [ expression1 ]

do

statement1

done

## for

for i in {0...100...2}

do

statement1

done

for (( i=0; i<5; i++ ))

do

statement1

done

array=(1 2 3)

for i in ${array[@]}

do

echo $i

echo 'hello'

done

# read

read line < “/dev/stdin”

# write

(1) ls -al 1>file1.txt 2>file2.txt

## 1 is the name of stdin

## 2 is the name of stderr

(2) ls -al >file1.txt

## by default, stdout

(3) ls -al >file1.txt 2>&1

ls -al >& file1.txt

redirect stderr to stdout

# nested scripts

script1.sh:

MESSAGE=”HELLO”

export MESSAGE # in script, like C++, making it global

./script2.sh

script2.sh:

echo $MESSAGE

# string operations

str1=”hello”

str2=”world”

if [ $str1 == $str2 ]

if [ $str1 \< $str2 ]

if [ $str1 \> $str2 ]

## concatenation

str3=$str1$str2

## lower case

${str1^}

${str1^l}

${str1^[ae]}

## upper case

${str1^^}

${str1^^l}

${str1^^[ae]}

# arithmetic operations

## $(())

n1=4

n2=5

echo $(( n1 + n2 ))

echo $(( n1 - n2 ))

echo $(( n1 \* n2 ))

echo $(( n1 / n2 ))

echo $(( n1 % n2 ))

## equivalent to:

$(expr )

$(expr $n1 + $n2)

$(expr $n1 - $n2)

$(expr $n1 \\* $n2)

$(expr $n1 / $n2)

$(expr $n1 % $n2)

## hex to decimal

$((16#AB))

# Function

## definition

function funcName()

{

echo “this is a new function”

}

## call a function

function func(){

local a=$1

echo "inside function $a"

}

echo $a

func 10

echo $a

## variable scope

local a = $1

otherwise

a is global variable

# File operators

|  |  |  |
| --- | --- | --- |
| **-b file** | Checks if file is a block special file; if yes, then the condition becomes true. | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true. | [ -k $file ] is false. |
| **-p file** | Checks if file is a named pipe; if yes, then the condition becomes true. | [ -p $file ] is false. |
| **-t file** | Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. | [ -t $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size greater than 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file exists; is true even if file is a directory but exists. | [ -e $file ] is true. |

# grep

grep [options] pattern [files…]

-i → ignore case

-n → include line number

-c → print count

-v → invert match ( lines that don’t match )

# sed

sed pattern filename

sed -i ‘s/2000/8000/g’ $filename

-i → in place

# How to debug shell script

1. bash -x ./script.sh

it shows step by step what is executed

2. #! /bin/bash -x

same as above

## specify what block is to be debugged

set -x

block to be debugged

set +x