Control Operators

Operators

we put more than one command on the command line using control operators.

| **Control Operator** | **Usage** |
| --- | --- |
| ; semicolon | More than one command can be used in a single line. |
| & ampersand | Command ends with & and doesn't wait for the command to finish. |
| $? dollar question mark | Used to store exit code of the previous command. |
| && double ampersand | Used as logical AND. |
| || double vertical bar | Used as logical OR. |
| Combining && and || | Used to write if then else structure in the command line. |
| # pound sign | Anything was written after # will be ignored. |

; semicolon

You can put two or more commands on the same line separated by a semicolon ; .

The shell will scan the line until it reaches the semicolon. All the arguments before this semicolon will be considered a separate command from all the arguments after the semicolon. Both series will be executed sequentially with the shell waiting for each command to finish before starting the next one.

user@clarusway:~$ echo Hello

Hello

user@clarusway:~$ echo World

World

user@clarusway:~$ echo Hello ; echo World

Hello

World

### & ampersand

When a line ends with an ampersand &, the shell will not wait for the command to finish. You will get your shell prompt back, and the command is executed in background. You will get a message when this command has finished executing in background.

user@clarusway:~$ sleep 20 &

[1] 14122

user@clarusway:~$

...wait 20 seconds...

user@clarusway:~$

[1]+ Done sleep 20

* Look at the above, command "sleep 20 &" has displayed a message after 20 seconds.
* Meanwhile, in the shell prompt, we can write any other command.

### $? dollar question mark

$? is a shell parameter and not a variable, since you cannot assign a value to $?.

user@clarusway~$ touch file1

user@clarusway~$ echo $?

0

user@clarusway~$ rm file1

user@clarusway~$ echo $?

0

user@clarusway~$ rm file1

rm: cannot remove `file1': No such file or directory

user@clarusway~$ echo $?

1

If status shows '0' then command was successfully executed and if shows '1' then command was a failure.

### && double ampersand

The shell will interpret && as a logical AND. When using && the second command is executed only if the first one succeeds (returns a zero exit status).

user@clarusway:~$ echo first && echo second

first

second

user@clarusway:~$ zecho first && echo second

-bash: zecho: command not found

user@clarusway:~$ cd gen && ls

file1 file3 File55 fileab FileAB fileabc

file2 File4 FileA Fileab fileab2

user@clarusway:/gen$ cd gen && ls

-bash: cd: gen: No such file or directory

### || double vertical bar

The || represents a logical OR. The second command is executed only when the first command fails (returns a non-zero exit status).

user@clarusway:~$ echo first || echo second ; echo third

first

third

user@clarusway:~$ zecho first || echo second ; echo third

-bash: zecho: command not found

second

third

Q: What is the difference between **|** and **||** in Linux?  
A: A pipe (**|**) is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing.  
But **||** is a logical OR operator. The second command is executed only when the first command fails.

 - Interview Q&A

combining && and ||

You can use this logical AND and logical OR to write an if-then-else structure on the command line. This example uses echo to display whether the rm command was successful.

user@clarusway:~$ rm file1 && echo It worked! || echo It

    failed!

It worked!

user@clarusway:~$ rm file1 && echo It worked! || echo It

    failed!

rm: cannot remove `file1': No such file or directory

It failed!

* If first condition (if) will be fulfilled then command line execution stops there.
* But if first condition is a failure, then second one (else) executes.

## Control Operators

### # pound sign

Everything written after a pound sign (#) is ignored by the shell. This is useful to write a shell comment but has no influence on the command execution or shell expansion.

user@clarusway:~$ mkdir test # we create a

    directory

user@clarusway:~$ cd test #### we enter

    the directory

user@clarusway:~$ ls # is it empty

    ?

### \ escaping special characters and end of line backslash

The backslash \ character enables the use of control characters, but without the shell interpreting it, this is called escaping characters.

user@clarusway:~$ echo hello \; world

hello ; world

user@clarusway:~$ echo hello\ \ \ world

hello world

user@clarusway:~$ echo escaping \\\ \#\ \&\ \"\ \'

escaping \ # & " '

user@clarusway:~$ echo escaping \\\?\\*\"\'

escaping \?\*"'

Lines ending in a backslash are continued on the next line. The shell does not interpret the newline character and will wait on shell expansion and execution of the command line until a newline without backslash is encountered.

user@clarusway:~$ echo This command line \

> is split in three \

> parts

This command line is split in three parts