Chapter 2c. control operators

In this chapter we put more than one command on the command line using **control operators**. We also briefly discuss related parameters ($?) and similar special characters(&).

2c.1. ; 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.

```
[clim@sop~]$ echo Hello

Hello

[clim@sop~]$ echo World

World

[clim@sop~]$ echo Hello ; echo World

Hello World

[clim@sop~]$
```

2c.2. & 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.

```
[clim@sop~]$ sleep 20 &

[1] 7925

[clim@sop~]$

...wait 20 seconds...
[clim@sop~]$

[1]+  Done                     sleep 20
```

The technical explanation of what happens in this case is explained in the chapter about **processes**.

2c.3. $? dollar question mark

The exit code of the previous command is stored in the shell variable **$?**. Actually **$?** is a shell parameter and not a variable, since you cannot assign a value to **$?**.

```
clim@sop:~/test$ touch file1
clim@sop:~/test$ echo $?

0

clim@sop:~/test$ rm file1
clim@sop:~/test$ echo $? 0

clim@sop:~/test$ rm file1

rm: cannot remove `file1': No such file or directory
clim@sop:~/test$ echo $?

1

clim@sop:~/test$
```

2c.4. && 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).

```
clim@sop:~$ echo first && echo second
first

second

clim@sop:~$ zecho first && echo second

-bash: zecho: command not found
```

Another example of the same **logical AND** principle. This example starts with a working **cd** followed by **ls**, then a non-working **cd** which is **not** followed by **ls**.

```
[clim@sop~]$ cd gen && ls

file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
[clim@sop bin]$ cd gen && ls

-bash: cd: bin: No such file or directory
```

2c.5. || 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).

```
clim@sop:~$ echo first || echo second ; echo third
first

third

clim@sop:~$ zecho first || echo second ; echo third

-bash: zecho: command not found
second

third
clim@sop:~$
```

Another example of the same **logical OR** principle.

```
[clim@sop~]$ cd gen || ls
[clim@sopgen]$ cd gen || ls

-bash: cd: gen: No such file or directory
file1  file3  File55  fileab  FileAB   fileabc
file2  File4  FileA   Fileab  fileab2
```

2c.6. 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.

```
clim@sop:~/test$ rm file1 && echo It worked! || echo It failed!
It worked!

clim@sop:~/test$ rm file1 && echo It worked! || echo It failed!
rm: cannot remove `file1': No such file or directory

It failed!
clim@sop:~/test$
```

2c.7. # 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.

```
clim@sop:~$ mkdir test        # we create a directory
clim@sop:~$ cd ./test
clim@sop:~/test$ ls           #### we enter the directory
clim@sop:~/test$
                              # is it empty ?
```

2c.8. \ escaping special characters

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

```
[clim@sop~]$ echo hello \; world

hello ; world

[clim@sop~]$ echo hello\ \ \ world

hello   world

[clim@sop~]$ echo escaping \\\ \#\ \&\ \"\ \'
escaping \ # & " '

[clim@sop~]$ echo escaping \\\?\*\"\'
escaping \?*"'
```

2c.8.1 end of line backslash

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.

```
[clim@sop ~]$ echo This command line \

> is split in three \

> parts

This command line is split in three parts
[clim@sop ~]$
```

2c.9. practice: control operators

0. Each question can be answered by one command line!

1. When you type **passwd**, which file is executed ?

2. What kind of file is that ?

3. Execute the **pwd** command twice. (remember 0.)

4. Execute **ls** after **cd /etc**, but only if **cd /etc** did not error.

5. Execute **cd /etc** after **cd etc**, but only if **cd etc** fails.

6. Echo **it worked** when **touch test42** works, and echo **it failed** when the **touch** failed. All on one command line as a normal user (not root). Test this line in your home directory and in **/bin/** .

7. Execute **sleep 6**, what is this command doing ?

8. Execute **sleep 200** in background (do not wait for it to finish).

9. Write a command line that executes **rm file55**. Your command line should print 'success' if file55 is removed, and print 'failed' if there was a problem.

   (optional)10. Use echo to display "Hello World with strange' characters \ * [} ~ \ \ ." (including all quotes) control