SISTEMI OPERATIVI: LABORATORIO

PROGETTO:

- **Sistema Operativo** da usare è da definire, ma probabilmente verrà usato **Ubuntu v18.04**.
- Il progetto consiste nel realizzare un **programma in C** che soddisfa dei requisiti (verranno specificati successivamente).
- I requisiti che il programma deve soddisfare saranno in una forma discorsiva (possiamo realizzarlo come vogliamo, basta che faccia quello che deve fare).
- Se il programma compila e funziona non è detto che si passi (guardano anche il modo in cui è stato realizzato) come se non compila o ci sono problemi, non è detto che non si passi il progetto:
    - C'è una **fase di discussione** (ci si può dividere i compiti, ma tutti devono sapere tutto del progetto).

COMANDI:

- CONCATENAZIONE DI COMANDI:
    - **pwd ; date**:
        - con il '**;**' vengono concatenati i comandi (viene eseguito pwd e una volta terminata la sua esecuzione, viene eseguito il comando date).
    - **pwd && date**:
        - anche in questo caso si ha una "*concatenazione*" di comandi, ma si ha un comportamento diverso rispetto a quello descritto sopra.
        - Visto che concatena con un **AND** logico, se il comando **pwd** viene eseguito con successo, allora anche il comando date viene eseguito.
        - Se il comando **pwd** non dovesse essere eseguito, neanche date viene eseguito (F && X = F)
    - **pwd || date**:
        - caso simile a quello dell'AND solo che si ha un **OR** logico tra i due comandi.
        - Se **pwd** non viene eseguito, **date** viene eseguito lo stesso

```
chabby@Chabby:/home$ pwd ; date
/home
Fri Feb 21 14:59:31 CET 2020
chabby@Chabby:/home$ xpwd && date

Command 'xpwd' not found, did you mean:

  command 'xwd' from deb x11-apps
  command 'xpad' from deb xpad
  command 'hpwd' from deb hfsutils
  command 'pwd' from deb coreutils

Try: sudo apt install <deb name>

chabby@Chabby:/home$ xpwd || date

Command 'xpwd' not found, did you mean:

  command 'xpad' from deb xpad
  command 'pwd' from deb coreutils
  command 'xwd' from deb x11-apps
  command 'hpwd' from deb hfsutils

Try: sudo apt install <deb name>

Fri Feb 21 14:59:51 CET 2020
```

- COMANDO PER CONTARE RIGHE/PAROLE:
    - **wc** (word count)
        - **ls /home | wc**:
            - **ls /home** genera dei dati (in questo caso elenca tutti gli elementi presenti nella cartella **/home**) e li invia sul canale di uscita che conosce (il comando **ls** non sa dove vanno i dati, lui sa solo che i dati in uscita vanno mandati in quel canale).
            - La **bash** pesca questi dati e li mostra a video.
            - Il carattere '**|**' fa si che i dati che il comando **ls** genera, invece che mandarli sulla rotta di default, li mandi al comando **wc**.
            - **wc** riceve questi dati, li elabora e anche lui manda i dati in uscita sul suo canale di output di default (poi la **bash** mostrerà questi dati).
            - Notare che l'output che viene mostrato è l'output dell'ultimo comando eseguito (non sempre si può inoltrare i dati generati da un comando ad un altro comando; quest'ultimo deve accettare dati in input).

```
chabby@Chabby:~$ ls /home/chabby/esempio.txt | wc
      1       1      25
```

- **wc < tmp/esempio.txt**:
  - viene letto il file **esempio.txt** e tutti i dati presi dal file vengono passati in input al comando **wc** (tramite una **pipe** vengono passati questi dati letti).
  - poi **wc** usando questi dati, genera l'output e lo inoltra sul suo canale di output (che verrà usato dalla **bash** per mostrare i dati a video).
  - il simbolo '**<**' viene usato per dire "prendi i dati dal file **esempio.txt**" e passali in input al comando **wc**.

```
chabby@Chabby:~$ wc < /home/chabby/esempio.txt
1 1 5
```

- NOTE:
  - I comandi hanno 3 canali:
    - **Input**: canale di entrata (tastiera)
    - **Output**: canale di uscita (video)
    - **Error**: usando questo canale, posso differenziare errori dai dati corretti

- COMANDO WHICH:
  - Il comando **which** mi restituisce il percorso dove è salvato il programma che sto cercando di / voglio eseguire(**path**).

```
chabby@Chabby:~$ which echo
/bin/echo
```

- BASH:
  - La **bash** è un programma che apre un **prompt** e aspetta che l'utente scriva "qualche cosa".
  - Quel "qualcosa" viene interpretato come comando e viene eseguito (se non riconosce il comando, la **bash** restituisce un errore).
  - Quando scrivo in comando, il programma **bash** lo cerca nella sua cartella e nei posti che conosce (se non è un comando interno) e lo esegue.

- COMANDO PER ELENCARE INFORMAZIONI SU **FILE** ED IL CONTENUTO DI **DIRECTORY**:
  - **ls**:
    - mostra informazioni ed il contenuto di file e cartelle

```
chabby@Chabby:~$ ls
esempio.txt  esempio1.sh  esempio2.sh  esempio3.sh
```

  - **ls *.sh**:
    - elenca tutti i file che hanno come estensione .sh

```
chabby@Chabby:~$ ls *.sh
esempio1.sh  esempio2.sh  esempio3.sh
```

  - **ls -l *.sh**:
    - restituisce i dati relativi ai file: tipo di file, nome del possessore, nome del gruppo, dimensione, ultima modifica e i permessi associati a quel file, divisi in 3 gruppi (proprietario, domain user e user):
      - **rw**: read & write (posso sia leggere che scrivere)
      - **r**: read (posso solo leggere)

```
-rw-r--r-- 1 chabby chabby 555 Feb 21 15:32 esempio1.sh
-rw-r--r-- 1 chabby chabby 410 Feb 21 15:33 esempio2.sh
-rw-r--rw- 1 chabby chabby 374 Feb 21 15:38 esempio3.sh
```

- COMANDO PER MODIFICARE I **PERMESSI** PER UN FILE:
  - **Opzioni**:
    - **a**: all | **g**: group (domain user) | **u**: user | **+**: aggiungo permesso | **-**: tolgo permesso
    - **r**: read | **w**: read | **x**: execution

  - **chmod g+w esempio1.sh**:
    - modifico i permessi del file esempio1.sh e modifico i permessi del domain user, che ora possono anche scrivere (oltre che leggere).

```
chabby@Chabby:~$ chmod g+w esempio1.sh
chabby@Chabby:~$ ls -l *.sh
-rw-rw-r-- 1 chabby chabby 555 Feb 21 15:32 esempio1.sh
-rw-r--r-- 1 chabby chabby 410 Feb 21 15:33 esempio2.sh
-rw-r--rw- 1 chabby chabby 374 Feb 21 15:38 esempio3.sh
```

  - **chmod +x esempio1.sh**:
    - aggiungo il permesso di esecuzione (**x**) per tutti gli utenti per il file **esempio1.sh**
    - sto dichiarando che il file **esempio1.sh** deve essere trattato come un eseguibile.

```
chabby@Chabby:~$ chmod +x esempio1.sh
chabby@Chabby:~$ ls -l *.sh
-rwxrwxr-x 1 chabby chabby 555 Feb 21 15:32 esempio1.sh
-rw-r--r-- 1 chabby chabby 410 Feb 21 15:33 esempio2.sh
-rw-r--rw- 1 chabby chabby 374 Feb 21 15:38 esempio3.sh
```

  - **chmod -x esempio1.sh**:
    - tolgo il permesso di esecuzione per il file **esempio1.sh**

```
chabby@Chabby:~$ chmod -x esempio1.sh
chabby@Chabby:~$ ls -l *.sh
-rw-rw-r-- 1 chabby chabby 555 Feb 21 15:32 esempio1.sh
-rw-r--r-- 1 chabby chabby 410 Feb 21 15:33 esempio2.sh
-rw-r--rw- 1 chabby chabby 374 Feb 21 15:38 esempio3.sh
```

  - **chmod u+x esempio1.sh**:
    - modifico i permessi solo per lo user (chi ha creato il file) e solo lui può eseguire il file **esempio.sh**

```
chabby@Chabby:~$ chmod u+x esempio1.sh
chabby@Chabby:~$ ls -l *.sh
-rwxrw-r-- 1 chabby chabby 555 Feb 21 15:32 esempio1.sh
-rw-r--r-- 1 chabby chabby 410 Feb 21 15:33 esempio2.sh
-rw-r--rw- 1 chabby chabby 374 Feb 21 15:38 esempio3.sh
```

- COMANDO PER ESEGUIRE:
  - Se provo ad eseguire il file, ottengo un errore, in quanto quando **bash** esegue un comando, non controlla mai la cartella corrente dove è situato.
  - Per far sì che venga eseguito devo scrivere l'intero percorso del file.
  - **./esempio1.sh:**

```
chabby@Chabby:~$ esempio1.sh
esempio1.sh: command not found
chabby@Chabby:~$ ./esempio1.sh
hello word
```

  - **bash esempio1.sh**:
    - passo a bash il file che lo esegue

```
chabby@Chabby:~$ bash esempio1.sh
hello word
```

- COMANDO PER OPERAZIONI MATEMATICHE:
  - **bc**:
    - accetta **espressioni matematiche** e le elabora.
    - prende in input l'espressione matematica, elabora e fa i suoi calcoli e successivamente stampa il risultato.

  - **echo '1.12 + 2.02' | bc**:
    - prima di stampare a video i dati, questi vengono passati a **bc** in input (operatore '**|**') che esegue la somma, restituisce il valore e questo viene stampato a video tramite il comando **echo**.

```
chabby@Chabby:~$ echo '1.12 + 2.02' | bc
3.14
```

- CODICE DI USCITA:
  - Tutti i comandi restituiscono un **codice di uscita (0-255)** al chiamante (al processo **bash**).
  - Ogni volta che viene eseguito un comando, **$?** viene **inizializzato** con il **codice di uscita** dell'**ultimo comando eseguito**.
  - **pwd ; echo $?** restuisce **0**, in quanto il comando **pwd** viene eseguito e ritorna al chiamante **0**, cioè "l'esecuzione è avvenuta con successo":

```
chabby@Chabby:~$ pwd ; echo $?
/home/chabby
0
```

  - **ls pippo ; echo $?**:
    - mi restituisce un errore, in quanto il comando pippo non esiste (**bash** non lo conosce e non lo ha tra i programmi) e ho un codice di errore **2**.
    - **ls** quando non trova un file ha come **codice di errore 2**.

```
chabby@Chabby:~$ ls pippo ; echo $?
ls: cannot access 'pippo': No such file or directory
2
```

  - Visto che **$?** contiene sempre il **codice di uscita dell'ultimo comando eseguito**, se io eseguo un comando (che non viene eseguito per qualsiasi problema) e poi faccio altro, nella variabile **$? non trovo più il codice di uscita di quel comando**, ma trovo il **codice di uscita del comando eseguito appena prima di $?**:

```
chabby@Chabby:~$ ls pippo ; echo "ciao" ; echo $?
ls: cannot access 'pippo': No such file or directory
ciao
0
```

- COMANDI PER IL CONFRONTO:
  - Vedi slide (sito docente o drive)

Esempio 1:

```
#!/bin/bash
#/usr/bin/env bash
#Questo è un commento, ma visto che c'è ! viene interpretato in maniera particolare
#Quando viene eseguito, visto che come prima riga abbiamo '!', questo commento viene
#utilizzato il riferimento /usr/bin/env bash per interpretare tutti gli altri comandi.
#Indica che questo script deve essere interpretato usando il programma che gli ho indicato
#nella prima riga (prende tutto lo script e lo passa al programma che gli ho passato).
#La prima riga indica a quale programma passare lo script

#mostro cartella corrente e data corrente
pwd ; date

#Script "Hello world"
echo "hello word";

#elenco i file nella cartella /tmp
ls /tmp
```

```
chabby@Chabby:~$ ./esempio1.sh
/home/chabby
Fri Feb 21 16:58:26 CET 2020
hello word
appInsights-nodeAIF-444c3af9-8e69-4462-ab49-4191e6ad1916
appInsights-nodeAIF-d9b70cd4-b9f9-4d70-929b-a071c400b217
hsperfdata_chabby
hsperfdata_root
npm-1700-7234b8c5
vscode-ipc-0bafc758-07f4-45be-9912-4af9eb6b7400.sock
vscode-ipc-1ca02267-6dd8-4d90-aa66-32c01e8d6fba.sock
vscode-ipc-3f49caa0-68ca-4688-845c-c5f847a4545d.sock
vscode-ipc-41cb742b-bb0b-4aa0-a9c2-986eaaa0d303.sock
vscode-ipc-46ffaffc-ad27-4edc-ace7-f5d5bb74398d.sock
vscode-ipc-4fe1404b-0507-43b1-bfa1-a6b447b63734.sock
vscode-ipc-7a7a5a3b-b5b6-4896-abda-0788fc32a02d.sock
vscode-ipc-84102568-ebdd-4052-bf47-177617a02ce1.sock
vscode-ipc-8c632b88-6f23-4ddc-9de7-488478ffdbb1.sock
vscode-ipc-b1b0c348-d4cf-4115-bd73-b4799272999b.sock
vscode-ipc-eaee931c-30ac-49b7-a836-6f05b1dd8ae2.sock
vscode-ipc-ed766f1f-9988-4635-b147-51c2d6e0a1a7.sock
vscode-ipc-f151e4a5-cb83-4ed5-b67f-0d5575dae2df.sock
vscode-typescript1000
```

Esempio 2:

```
#!/bin/bash

# x = 6 non va bene mettere spazi
# Tutto quello che scrivo viene visto come stringa (dipende dal contesto in cui vengono usate)
x=7

# $x viene usato per accedere alla variabile
# ${x} usato per accedere alla variabile x in modo più "sicuro"
echo Ciao $x !

# Quando viene eseguito, interpreta l'intera stringa come variabile, non vedo 7
echo Ciao $xilofono !

# Stampa 7ilofono
echo Ciao ${x}ilofono !
```

```
chabby@Chabby:~$ ./esempio2.sh
Ciao 7 !
Ciao !
Ciao 7ilofono !
```

Esempio 3:

```
#!/bin/bash

x=7

# Per eseguire si usa il ./[file]
# ./eg3.sh --help --esempio
# Tutto quello che viene scritto dopo il file, viene assegnato a $1, $2, ecc..
# In questo caso abbiamo che $1 = --help e $2 = --esempio
# $@ è l'insieme di tutti gli elementi ($1,...,$9)
# shift elimina il primo paramentro e sposta tutti gli altri
echo Dollaro 1 = $1
echo Dollaro 2 = $2
```

```
chabby@Chabby:~$ ./esempio3.sh --help --esempio
Dollaro 1 = --help
Dollaro 2 = --esempio
```

Esempio 4:

```
#!/bin/bash

x=2+3
echo "x=$x" #la vede come stringa, non stampa 5

z1=$(( x*2 )) #elabora il contenuto interno come espressione matematica (5*2)
z2=$(( $x*2 )) #il $x ha priorità suò $((..)) quindi abbiamo (2+3*2)

echo "z1=$z1"
echo "z2=$z2"

#manda la stringa a bc che stampa 3.14
echo '1.12 + 2.02' | bc

# da errore, perchè cerca 3.14 come comando ma visto che ho messo
# Con $() viene invocato un altro programma bash (un altro processo) e l'intera riga viene sostituita
# con il valore appena calcolato al posto di $(), quindi da errore perchè lo interpreta come comando
$(echo '1.12 + 2.02' | bc)

# Corretto
echo "$(echo '1.12 + 2.02' | bc)"
```

```
chabby@Chabby:~$ ./esempio4.sh
x=2+3
z1=10
z2=8
3.14
./esempio4.sh: line 18: 3.14: command not found
3.14
```

Esempio 5:

```
#!/bin/bash

#codice per testare condizioni
echo "res=$?"
x=3
#test è un comando che fa confronti e restituisce 0 se è andato a buon fine o 1 se non è andato bene
test $x -eq 3 #verifica se x è uguale a 3
echo "res=$?"

#comando [  ], posso leggerle come "BEGIN TEST" e "CLOSE TEST" (spazi importanti)
#verifica se x è uguale a 0
[ $x -eq 0 ] #notare gli spazi
echo "res=$?"

#if e then sono 2 comandi diversi, devo mettere il ; oppure vado a capo
#comandi in righe diverse vengono concatenati
if [[ $x>5 ]]; then
    echo "vero"
else
    echo "falso"
fi #fine if

#versione alternativa
if [[ $x>5 ]]; then echo "vero" ; else echo "falso" ; fi
```

```
chabby@Chabby:~$ ./esempio5.sh
res=0
res=0
res=1
falso
falso
```

Esempio 6:

```
#!/bin/bash

echo "Digita una lettera e premi INVIO"
read char #aspetta un comando prima di andare avanti
case $char in
    y|Y)
        echo "Input: y"
    ;; #alla fine devo mettere un doppio ;

    n)
        echo "Input: n"
    ;;

    *)
        echo "Input: ne y ne n"
    ;; #il * è un wildcat, se nessun caso si verifica stampa questo
esac #fine switch
```

```
chabby@Chabby:~$ ./esempio6.sh
Digita una lettera e premi INVIO
ciao
Input: ne y ne n
chabby@Chabby:~$ ./esempio6.sh
Digita una lettera e premi INVIO
n
Input: n
chabby@Chabby:~$ ./esempio6.sh
Digita una lettera e premi INVIO
y
Input: y
```