Terminale avanzato

Samuele Musiani, Alice Benatti

Università di Bologna, corso di Laurea in Informatica

23 novembre 2023

Un mare di shell

Cos'è una shell

Una `shell` è un programma che permette di parlare con il sistema operativo attraverso dei comandi da tastiera.

Cos'è una shell

Una `shell` è un programma che permette di parlare con il sistema operativo attraverso dei comandi da tastiera.

```
[samu@leibniz ~]$
```

Figura: Esempio di un *shell prompt*

Cos'è una shell

Esistono varie shell:

- sh
- bash
- zsh
- fish

Tutte servono per interagire con il sistema operativo, però sono tutte leggermente diverse sotto alcuni aspetti.

Di solito la predefinita nei sistemi Linux è bash.

I primi passi in un terminale

Aprire un terminale

Per utilizzare una `shell` è necessario disporre di un emulatore di terminale.

Un emulatore di terminale è un'applicazione che permette di interagire con la `shell`.

Esisto vari tipi di emulatori che offrono configurazioni e opzioni diverse:

- Alacritty
- Foot
- Kitty
- Konsole
- ...

Pipe e ridirezione: il potere della shell

Il potere della shell

Di solito si utilizzano comandi singoli. Nonostante molti di essi siano utili anche da soli, sono sicuramente più utili usati in combinazione con altri comandi.

Come si fa però a combinare più comandi?

Il potere della shell

Di solito si utilizzano comandi singoli. Nonostante molti di essi siano utili anche da soli, sono sicuramente più utili usati in combinazione con altri comandi.

Come si fa però a combinare più comandi? Ricordiamoci che:

- Un comando restituisce sempre[1] qualcosa sullo standard output

[1] In realtà non tutti i comandi restituiscono per forza qualcosa, es. `cd`

Il potere della shell

Di solito si utilizzano comandi singoli. Nonostante molti di essi siano utili anche da soli, sono sicuramente più utili usati in combinazione con altri comandi.

Come si fa però a combinare più comandi? Ricordiamoci che:

- Un comando restituisce sempre[1] qualcosa sullo standard output
- Lo standard output è considerato come un file (in memoria) dal sistema

[1] In realtà non tutti i comandi restituiscono per forza qualcosa, es. `cd`

Il potere della shell

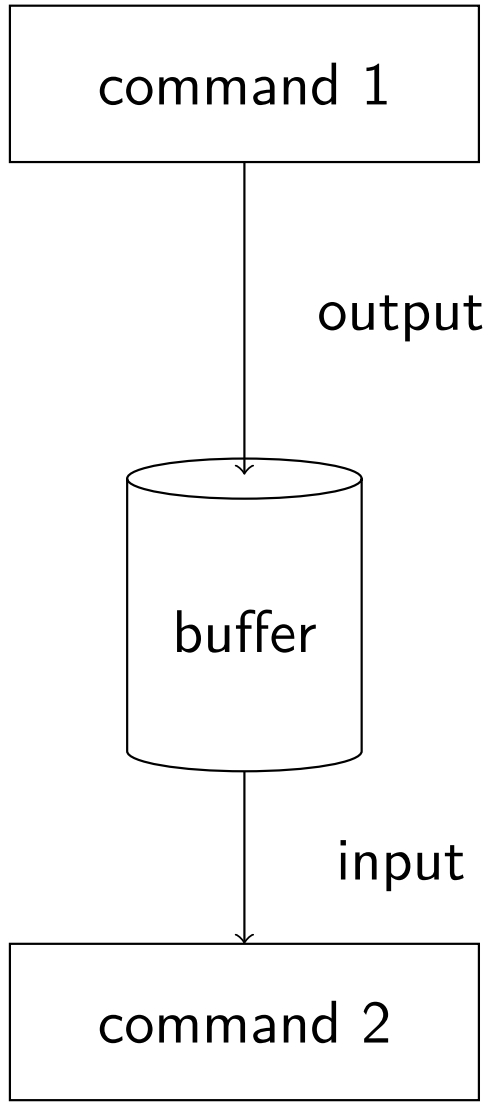
Di solito si utilizzano comandi singoli. Nonostante molti di essi siano utili anche da soli, sono sicuramente più utili usati in combinazione con altri comandi.

Come si fa però a combinare più comandi? Ricordiamoci che:

- Un comando restituisce sempre[1] qualcosa sullo standard output
- Lo standard output è considerato come un file (in memoria) dal sistema
- La maggior parte dei comandi visti fino ad adesso hanno la possibilità di prendere in input lo standard output invece che un file classico

[1] In realtà non tutti i comandi restituiscono per forza qualcosa, es. `cd`

Il potere della shell



Pipe

Per prendere l'output di un comando e riderizionarlo in input verso un altro comando si usa la **pipe** |

Per esempio se vogliamo vedere tutti i file presenti in `/bin` il nostro terminale si riempie di scritte.

Possiamo visualizzare il lungo output con il comando `less`:`ls /bin | less`

Pipe - grep

Come abbiamo visto la lista di file presenti in `/bin` è molto lunga.
Se volessimo trovarne uno specifico?

Pipe - grep

Come abbiamo visto la lista di file presenti in `/bin` è molto lunga. Se volessimo trovarne uno specifico?

Nonostante esiste un comando apposito per cercare file, possiamo fare: `ls /bin | grep "firefox"` dove al posto di *firefox* può andarci una qualsiasi stringa.

Pipe - esempi

Di seguito una lista di esempi di utilizzo della `pipe`:

- `cat file1 file2 | grep "word"` cerca una stringa in più file
- `ls /bin | wc -l` conta quanti programmi sono presenti in `/bin`
- `ls /bin | grep "zip" | wc -l` conta quanti programmi hanno la stringa "*zip*" al loro interno nella cartella `/bin`
- `grep "castoro" animali | wc -l` conta le occorrenze di `castoro` trovate nel file `animali`
- `grep "the" book | less` mostra le occorrenze di `the` trovate in `book` attraverso il lettore `less`

Ridirezione su file

Come abbiamo visto è possibile mandare l'output di un comando nell'input di un altro comando.

Se volessimo salvare l'output di un comando su un file?

Ridirezione su file

Come abbiamo visto è possibile mandare l'output di un comando nell'input di un altro comando.

Se volessimo salvare l'output di un comando su un file?
Esiste l'**operatore di ridirezione** >

Al posto di indirizzare l'output in un comando, scrive direttamente su un file.

Sintassi: `comando > file`

ATTENZIONE: Alla *shell* non interessa se il file esiste già, quindi se esiste lo SOVRASCRIVE COMPLETAMENTE[2].

[2] In reltà in shell più moderne tipo zsh esce un prompt se il file esiste già

Ridirezione su file non distruttiva

Esiste anche un operatore per indirizzare su file l'output di un comando senza sovrascrivere il contenuto del file, ma "appendendo" alla fine del file il contenuto scritto.

Sintassi: `comando >> file`

Si usa nello stesso modo dell'operatore classico

Ridirezione dello standard error

Di predefinito l'operatore di redirezione, come la pipe, reindirizza solo lo standard output. Questo significa che se un comando restituisce un errore questo non verrà riportato nel file indicato:

```
[samu@leibniz ~]$ ls /asdkjc > /tmp/castoro
ls: cannot access '/asdkjc': No such file or
directory
[samu@leibniz ~]$
```

Figura: Esempio di stderr non ridirezionato

Per ridirezionare anche lo standard error dentro il file è necessario specificarlo alla shell.

Ridirezione dello standard error

Lo standard error in linux ha come *file descriptor* il **2**. Non importa sapere cosa sia un file descriptor, basta ricordarsi che **stdout = 1** e **stderr = 2**.

Per ridirezionare lo stderr sullo stdout si usa la seguente sintassi: **2>&1**

Redirezionando lo standard error verso lo standard output è possibile scrivere entrambi su file:

```
[samu@leibniz ~]$ ls /asdkjc > /tmp/test 2>&1
[samu@leibniz ~]$ cat /tmp/test
ls: cannot access '/asdkjc': No such file or
directory
```

Figura: Esempio di stderr ridirezionato

Ridirezione dello standard error

Da notare che questo funziona anche per la pipe:

```
[samu@leibniz ~]$ ls /asdkjc | head -c 10
ls: cannot access '/asdkjc': No such file or
directory
[samu@leibniz ~]$ ls /asdkjc 2>&1 | head -c 10
ls: cannot
```

Figura: Esempio di stderr ridirezionato

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin
- uniq: Se ci sono più linee consecutive uguali ne lascia solo una

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin
- uniq: Se ci sono più linee consecutive uguali ne lascia solo una
- grep: Cerca il testo passato come argomento

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin
- uniq: Se ci sono più linee consecutive uguali ne lascia solo una
- grep: Cerca il testo passato come argomento
- wc: Conta linee, parole e carateri passati in input

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin
- uniq: Se ci sono più linee consecutive uguali ne lascia solo una
- grep: Cerca il testo passato come argomento
- wc: Conta linee, parole e carateri passati in input
- head: Manda in stdout solo la prima parte del testo

Filtri

Quando si combinano assieme più comandi spesso i seguenti tornano utili:

- sort: Restituisce in stdout il testo preso in stdin
- uniq: Se ci sono più linee consecutive uguali ne lascia solo una
- grep: Cerca il testo passato come argomento
- wc: Conta linee, parole e carateri passati in input
- head: Manda in stdout solo la prima parte del testo
- tail: Manda in stdout solo l'ultima parte del testo

/dev/null

Come si fa a ridirezionare soltanto lo stderr su file e "buttare via" lo stdout?

/dev/null

Come si fa a ridirezionare soltanto lo stderr su file e "buttare via" lo stdout?

Esiste in Linux un "file" che indipendentemente da cosa gli passate lo butta via. Questo "file" è **/dev/null**.

```
[samu@leibniz ~]$ ls /asdkjc / 2> /tmp/pippo
1> /dev/null
[samu@leibniz ~]$ cat /tmp/pippo
ls: cannot access '/asdkjc': No such file or
directory
```

Figura: Esempio di stderr ridirezionato

Ricordiamoci che stdout = 1 e stderr = 2.

echo, echo, echo, echo...

Echo, echo, echo...

Il comando `echo`, come suggerisce il suo nome, serve per stampare quello che gli viene passato come argomento:

```
[samu@leibniz ~]$ echo "ciao"
ciao
[samu@leibniz ~]$ echo "Questa è una frase"
Questa è una frase
```

Figura: Esempio di echo

A prima vista può sembra abbastanza inutile, in realtà ha tantissime applicazioni.

Seeing the world as the shell sees it

```
[samu@leibniz ~]$ echo *
Documents Downloads go Pictures VirtualBox
[samu@leibniz ~]$ echo /usr/*/share
/user/kerberos/share /user/local/share
```

Figura: Esempio di echo con una wildcard

Come possiamo vedere echo non stampa l'asterisco come potremmo aspettarci. Lo tratta invece come la shell: ovvero come una wildcard.

Seeing the world as the shell sees it

```
[samu@leibniz ~]$ echo *
Documents Downloads go Pictures VirtualBox
[samu@leibniz ~]$ echo /usr/*/share
/user/kerberos/share /user/local/share
```

Figura: Esempio di echo con una wildcard

Come possiamo vedere echo non stampa l'asterisco come potremmo aspettarci. Lo tratta invece come la shell: ovvero come una wildcard.

In realtà è proprio la shell a sostituire la wildcard con le entries opportune prima di eseguire il comando `echo`.

Seeing the world as the shell sees it

Esiste una particolare sintassi per indicare un range in bash:

```
[samu@leibniz ~]$ echo {1..11}
1 2 3 4 5 6 7 8 9 10 11
[samu@leibniz ~]$ echo {c..t}
c d e f g h i j k l m n o p q r s t
```

Figura: Esempio di echo range

La shell, prima di esegure l'echo, sostituisce le parentesi graffe con il range specificato.

Seeing the world as the shell sees it

Esiste una particolare sintassi per indicare un range in bash:

```
[samu@leibniz ~]$ echo {1..11}
1 2 3 4 5 6 7 8 9 10 11
[samu@leibniz ~]$ echo {c..t}
c d e f g h i j k l m n o p q r s t
```

Figura: Esempio di echo range

La shell, prima di esegure l'echo, sostituisce le parentesi graffe con il range specificato.

Non funziona solo con echo in quanto è la shell stessa a fare la sostituzione. Funziona molto bene con `mkdir`:

```
mkdir {2020..2021}-{01..12}-{01..31}
```

Sostituzione di comandi

La pipe non riesce a fare tutto

Con la pipe possiamo combinare più comandi assieme, però non riusciamo a fare ancora tutto.

Se volessimo vedere il tipo di tutti i file che hanno la parola zip nel nome dentro la cartella `/bin` come facciamo?

La pipe non riesce a fare tutto

Con la pipe possiamo combinare più comandi assieme, però non riusciamo a fare ancora tutto.

Se volessimo vedere il tipo di tutti i file che hanno la parola zip nel nome dentro la cartella `/bin` come facciamo?

```
$ ls -d /bin/* | grep zip | file
```

Figura: Esempio SBAGLIATO. Il prompt è stato tralasciato per questioni di spazio. Al suo posto è utilizzato soltanto il dollaro

Questo comando restituisce un errore perché il comando `file` non riesce ad interpretare lo standard input tramite la pipe.

La pipe non riesce a fare tutto

Con la pipe possiamo combinare più comandi assieme, però non riusciamo a fare ancora tutto.

Se volessimo vedere il tipo di tutti i file che hanno la parola zip nel nome dentro la cartella `/bin` come facciamo?

```
$ ls -d /bin/* | grep zip | file
```

Figura: Esempio SBAGLIATO. Il prompt è stato tralasciato per questioni di spazio. Al suo posto è utilizzato soltanto il dollaro

Questo comando restituisce un errore perché il comando `file` non riesce ad interpretare lo standard input tramite la pipe.
Il comando file vuole gli input come argomenti: `file ...`

Sostituzione di comandi

La shell ammette una sintassi per la sostituzioni di comandi all'interno di un'espressione.

Sintassi: `$(comando)`

La shell sostituirà l'intera espressione al risulto del comando:

```
[samu@leibniz ~]$ echo $(ls)
Documents Downloads go Pictures VirtualBox
[samu@leibniz ~]$ ls
Documents Downloads go Pictures VirtualBox
```

Figura: Esempio di sostituzione del comando `ls`

Sostituzione di comandi

Ritornando a quello che volevamo fare prima: *Se volessimo vedere il tipo di tutti i file che hanno la parola zip nel nome dentro la cartella `/bin` come facciamo?*

Sostituzione di comandi

Ritornando a quello che volevamo fare prima: *Se volessimo vedere il tipo di tutti i file che hanno la parola zip nel nome dentro la cartella `/bin` come facciamo?* Con la sostituzione di comandi!

```
$ file $(ls -d /bin/* | grep zip)
```

Figura: Esempio di più comandi. Il prompt è stato tralasciato. Al suo posto è utilizzato un dollaro (il primo).

Matematica base

Si possono fare anche operazioni matematiche con questo tipo di sostituzione:

```
[samu@leibniz ~]$ echo $((1 + 1))
2
[samu@leibniz ~]$ echo $((9 * 5))
45
[samu@leibniz ~]$ echo 3GB are $((3 * 1024))MB
3GB are 3072MB
```

Figura: Esempio di più comandi. Il prompt è stato tralasciato. Al suo posto è utilizzato un dollaro (il primo).

Variabili

Variabili

È possibile salvare dei valori in delle **variabili**.

Alcuni programmi sono configurabili attraverso le variabili di ambiente.

Esistono delle variabili già esistenti nella shell che possono essere viste con il comando `echo`.

```
[samu@leibniz ~]$ echo $USER
samu
[samu@leibniz ~]$ echo $SHELL
/bin/bash
```

Figura: Esempio variabile

Le variabili saranno più utili negli script che nella shell direttamente.

Assegnamento delle variabili

Per assegnare un valore ad una variabile si può usare la seguente sintassi: `varname=content`.

```
[samu@leibniz ~]$ pippo=cane
[samu@leibniz ~]$ echo $pippo
cane
```

Figura: Esempio di assegnamento variabile

Una variabile è vuota se quando si printa con `echo` non ha nulla al suo interno:

```
[samu@leibniz ~]$ echo $papera

[samu@leibniz ~]$
```

Figura: Esempio di variabile vuota

Uso di variabili

Si possono utilizzare le variabili nei comandi:

```
[samu@leibniz ~]$ src=/home/samu/Immagini
[samu@leibniz ~]$ dest=/tmp/immagini
[samu@leibniz ~]$ cp -r $src $dest
```

Figura: Esempio di copia utilizzando variabili

```
[samu@leibniz ~]$ programs=$(ls /bin)
[samu@leibniz ~]$ echo $programs | less
```

Figura: Tutti i programmi in `/bin` sono salvati nella variabile *programs*

La variabile path

Una variabile spesso usata è la variabile di **PATH**.

Quando digitato un comando la shell andrà a cercare se esiste un eseguibile con lo stesso nome del comando nelle directory indicate dalla variabile `PATH`.

```
[samu@leibniz ~]$ echo $PATH
/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/l
ib/jvm/default/bin:/usr/bin/site_perl
[samu@leibniz ~]$
```

Figura: Percorsi nella variabile `PATH`

Export

Di predefinito le variabili non sono propagate anche sui processi figli della shell.

È quindi spesso necessario esportarle per renderle visibili.

```
[samu@leibniz ~]$ pippo=cane
[samu@leibniz ~]$ echo $pippo
cane
[samu@leibniz ~]$ bash -c 'echo $pippo'

[samu@leibniz ~]$ export pippo=cane
[samu@leibniz ~]$ bash -c 'echo $pippo'
cane
```

Figura: Percorsi nella variabile `PATH`

Processi

htop

Linux è sistema che permette l'esecuzione di più processi simultaneamente.

Per vedere i processi in esecuzione (e altre informazioni) è molto comodo il comando `top` o `htop` (la versione nuova).

Se si vuole invece aver uno snapshot dei processi in esecuzione si può utilizzare il comando `ps`.

Per vedere tutti i processi su `ps` la flag è `-e`.

Pid e kill

Per identificare in modo univoco un processo si utilizza un intero positivo chiamato **pid**: Process IDentifier.

Pid e kill

Per identificare in modo univoco un processo si utilizza un intero positivo chiamato **pid**: Process IDentifier.

Se per caso volessimo terminare un processo lo si può fare attraverso il comando `kill` passando il pid del processo da uccidere.

Il pid può essere ottenuto attraverso il comando `ps`:

```
[samu@leibniz ~]$ ps
    PID TTY          TIME CMD
   8239 pts/3    00:00:03 zsh
  20813 pts/3    00:00:00 ps
[samu@leibniz ~]$ kill 8239
```

Figura: Esempio di `kill`

Bash scripting

Bash scripting

Spesso si vogliono automatizzare dei comandi della shell, oppure eseguirne più in "una volta sola".

Bash scripting

Spesso si vogliono automatizzare dei comandi della shell, oppure eseguirne più in "una volta sola".

La shell (e in particolare bash) permette di scrivere degli script.

Uno script è un file di test con un elenco di comandi che vengono eseguiti in sequenza da bash[3] .

Per specificare che interprete da usare è necessario indicare nella prima riga del file il percorso assoluto all'eseguibile della shell:

```
#!/bin/bash
```

Figura: Riga per indicare bash come interprete

[3] In realtà dall'interprete che specificate

Esempio di script

```
#!/bin/bash
echo "Reading programs"
PROG=$(ls -d /bin/*)
NUM=$(echo "$PROG" | grep "ca" | wc -l)
echo "Numero di programmi con 'ca' è $NUM"
```

Figura: Esempio di bash script

Questo script fa le seguenti operazioni:

1. Stampa sullo standar output "*Reading programs*"
2. Salva tutti i programmi presenti in `/bin` nella variabile `PROG`.
3. Salva nella variabile `NUM` il numero di tutte le stringhe presenti nella variabile `PROG` che hanno al loro interno "ca".
4. Stampa il valore presente nella variabile `NUM`.

Eseguire uno script

Per eseguire uno script bash è importante avere un file (lo script) con i comandi al suo interno.

La convenzione è che gli script bash si salvano con estenzione **.sh**

Inoltre bisogna renderlo eseguibile aggiungendogli il permesso:

`chmod +x script.sh`

Per eseguilo basta quindi indicare il suo percorso assoluto o relativo:

```
[samu@leibniz scripts]$ ls -al
-rw-r--r-- 1 samu samu Nov 20 17:17 script.sh
[samu@leibniz ~]$ chmod +x script.sh
-rwxr-xr-x 1 samu samu Nov 20 17:17 script.sh
[samu@leibniz ~]$ ./script.sh
```

Figura: Esecuzione di uno script bash

if, for, while

Non abbiamo il tempo di trattarli ma in bash esistono anche **if**, **for** e **while**:

```
#!/bin/bash
DEST="/mnt/usb/imgs"
IMGS=$(find ~/Images -type f -name "*.jpg")
a=1
for i in $IMGS; do
  echo "Coping $i to $DEST/$a.jpg"
  cp "$i" "$DEST/$a.jpg"
    a=$(($a+1))
done
```

Figura: Esempio di bash script

Una veloce introduzione a vim

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

- Quando si programma la maggior parte del tempo è passato a *modificare* il codice, non a scriverlo

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

- Quando si programma la maggior parte del tempo è passato a *modificare* il codice, non a scriverlo
- Modificare il codice include molto altro oltre a scrivere: eliminare, sostituire, riordinare, duplicare, formattare, ecc.

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

- Quando si programma la maggior parte del tempo è passato a *modificare* il codice, non a scriverlo
- Modificare il codice include molto altro oltre a scrivere: eliminare, sostituire, riordinare, duplicare, formattare, ecc.
- Ha senso facilitare tutta la parte di modifica del codice, più che di scrittura effettiva

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

- Quando si programma la maggior parte del tempo è passato a *modificare* il codice, non a scriverlo
- Modificare il codice include molto altro oltre a scrivere: eliminare, sostituire, riordinare, duplicare, formattare, ecc.
- Ha senso facilitare tutta la parte di modifica del codice, più che di scrittura effettiva
- Il mouse è una perdita di tempo quando si deve scrivere, se si può fare tutto da tastiera in modo efficiente è meglio

Un po' di contesto

Su Linux esistono due principali editor di testo: nano e vim.

Nano è un editor semplice che ha i comandi scritti a schermo per evitare di scordarseli.

La filosofia di vim invece è diversa. Facciamo qualche osservazione:

- Quando si programma la maggior parte del tempo è passato a *modificare* il codice, non a scriverlo
- Modificare il codice include molto altro oltre a scrivere: eliminare, sostituire, riordinare, duplicare, formattare, ecc.
- Ha senso facilitare tutta la parte di modifica del codice, più che di scrittura effettiva
- Il mouse è una perdita di tempo quando si deve scrivere, se si può fare tutto da tastiera in modo efficiente è meglio
- Un'operazione usa il minor numero di tasti possibili.

Modalità

In un editor classico ci si trova sempre nella modalità di *modifca*, ovvero se si schiacciano delle lettere vengono inserite nel file che si sta modificando.

Modalità

In un editor classico ci si trova sempre nella modalità di *modifca*, ovvero se si schiacciano delle lettere vengono inserite nel file che si sta modificando.

Per la maggir parte degli editor (compreso nano) questa è l'unica modalità presente.

Modalità

In un editor classico ci si trova sempre nella modalità di *modifca*, ovvero se si schiacciano delle lettere vengono inserite nel file che si sta modificando.

Per la maggir parte degli editor (compreso nano) questa è l'unica modalità presente.

Vim introduce tre modalità principali:

- Normal mode
- Insert mode
- Visual mode

Modalità

Vim introduce tre modalità principali:

- **Normal mode**: Modalità predefinita di vim. La si attiva con il tasto `ESC`. Permette di muoversi all'interno del documento in modo rapido. Di eliminare, copiare, incollare e spostare parti di testo.

Modalità

Vim introduce tre modalità principali:

- **Normal mode**: Modalità predefinita di vim. La si attiva con il tasto `ESC`. Permette di muoversi all'interno del documento in modo rapido. Di eliminare, copiare, incollare e spostare parti di testo.
- **Insert mode**: Modalità classica per l'inserimento e la scrittura di testo. È la stessa che si trova nella maggior parte degli editor. Ci si accede principalmente usando il tasto 'i', ma anche 'I', 'a', 'A', ecc.

Modalità

Vim introduce tre modalità principali:

- **Normal mode**: Modalità predefinita di vim. La si attiva con il tasto `ESC`. Permette di muoversi all'interno del documento in modo rapido. Di eliminare, copiare, incollare e spostare parti di testo.
- **Insert mode**: Modalità classica per l'inserimento e la scrittura di testo. È la stessa che si trova nella maggior parte degli editor. Ci si accede principalmente usando il tasto 'i', ma anche 'I', 'a', 'A', ecc.
- **Visual mode**: Permette di selezionare parti di testo, sia verticalmente che orizzontalmente. Ci si accede usanto il tato 'v'.

Normal mode

La **normal mode** è la predefinita e qulla su cui si dovrebbe sempre stare se non si necessitano di altre modalità. Per entrarci usare il tasto `ESC`.

Per spostarsi all'interno del testo in normal mode si usano le seguenti lettere e NON le freccette:

- j: Per spostarsi verso il basso di una riga
- k: Per spostarsi verso l'alto di una riga
- h: Per spostarsi a sinistra di un carattere
- l: Per spostarsi a destra di un carattere

Il motivo per cui si utilizzano queste lettere e non le freccette è il fatto che lo spostamento della mando per raggiungere le freccette è troppo grande e richiederebbe troppo tempo per la filosofia di vim.

Normal mode

Altre combinazioni di tasti base in normal mode:

- w: si sposta all'inizio della parola successiva
- e: si sposta alla fine della parola successiva
- b: si sposta all'inizio della parola precedente
- gg: si sposta all'inizio del file
- G: si sposta alla fine del file
- dd: cancella la riga su cui si trova il cursore
- yy: copia la riga su cui si trova il cursore
- p: incolla il testo copiato nella riga successiva al cursore
- x: elimina il carattere sotto il cursore
- u: annulla l'ultima operazione fatta
- ctrl + r: ripete l'ultima operazione annullata (il contrario di 'u')

Insert mode

Si utilizza la **insert mode** quando si vuole scrivere del testo. Per entrarci dalla normal mode si possono usare i seguenti tasti:

- i: si inizia a digitare prima della lettera sotto il cursore
- a: si inizia a digitare dopo della lettera sotto il cursore
- I: si inizia a digitare all'inizio della riga
- A: si inizia a digitare alla fine della riga
- o: si inizia a digitare in una nuova linea sottostante
- O: si inizia a digitare in una nuova linea sovrastante
- cc: si cancella la riga su cui si trova il cursore e si inizia a scrivere al suo posto

How to exit vim

Uscire da vim è una delle operazioni più complesse.

How to exit vim

Uscire da vim è una delle operazioni più complesse.

Molti dicono che l'unico modo sia riavviare il pc.

How to exit vim

Uscire da vim è una delle operazioni più complesse.

Molti dicono che l'unico modo sia riavviare il pc.

In realtà basta sapere come funziona!

Per uscire da vim bisogna trovarsi in normal mode e digitare **:q** e successivamente premere invio.

Se il file è stato modificato dall'apertura il comando non funzionerà perché vim non sa se vuoi salvare il file o no. Per salvarlo (ed uscire) si usa **:wq** e per non salvarlo (ed uscire) **:q!**

Per salvare quindi un file senza uscire basta **:w**

More work to do

More work to do

Sia bash che vim sono programmi immensi che non si finisce mai di conoscerli a fondo. Di seguito sono riportati alcuni comandi che potete approfondire:

- awk, sed, tr
- diff
- ip e ping
- rsync
- killall
- dd
- du, df

ATTENZIONE: Mai usare un comando prima di sapere cosa fa. Alcuni di questi possono anche essere pericolosi se usati senza un minimo di criterio.