

# Bash Basics

---

Università di Modena e Reggio Emilia

*Prof. Nicola Bicocchi ([nicola.bicocchi@unimore.it](mailto:nicola.bicocchi@unimore.it))*



# Utilità

---

# Builtins

---

Bash, come **interprete dei comandi**, esegue il contenuto di file binari presenti nel filesystem (es. /bin/ls)

```
$ ls  
$ which ls  
/bin/ls  
$ which which  
/usr/bin/which
```

# Builtins

---

Esistono particolari comandi, detti **builtins**, che non provengono dall'esecuzione di un file binario ma sono **implementati all'interno della shell**. Nel loro caso, *\$ which comando* non ritorna un percorso perchè il file binario non esiste! Ad esempio:

```
$ cd  
$ alias  
$ history  
$ logout
```

[https://www.gnu.org/software/bash/manual/html\\_node/Bash-Builtins.html](https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html)

# alias

---

```
$ alias ll='ls -l'  
$ ll  
drwxr-xr-x+ 51 nicola staff 1632 Mar 14 11:09 .  
drwxr-xr-x 6 root admin 192 Nov 14 2018 ..  
drwx-----@ 6 nicola staff 192 Feb 24 16:09 Applications  
drwx-----+ 6 nicola staff 192 Mar 19 00:10 Desktop  
drwx-----+ 7 nicola staff 224 Feb 29 12:15 Documents  
drwxr-xr-x+ 28 nicola staff 896 Mar 18 18:55 Downloads  
drwx-----@ 9 nicola staff 288 Mar 11 13:16 Dropbox  
drwx-----@ 10 nicola staff 320 Mar 14 11:10 Google Drive  
drwx-----@ 75 nicola staff 2400 Mar 3 19:07 Library  
drwx-----+ 3 nicola staff 96 Jul 6 2018 Movies  
$ unalias ll
```



# history

---

```
$ history
```

```
1 uname -a
```

```
2 clear
```

```
3 exit
```

```
4 ls
```

```
$ !!      (esegue ultimo comando)
```

```
$ !2      (esegue comando #2)
```

# Freccia su-giù, ctrl-r, tab

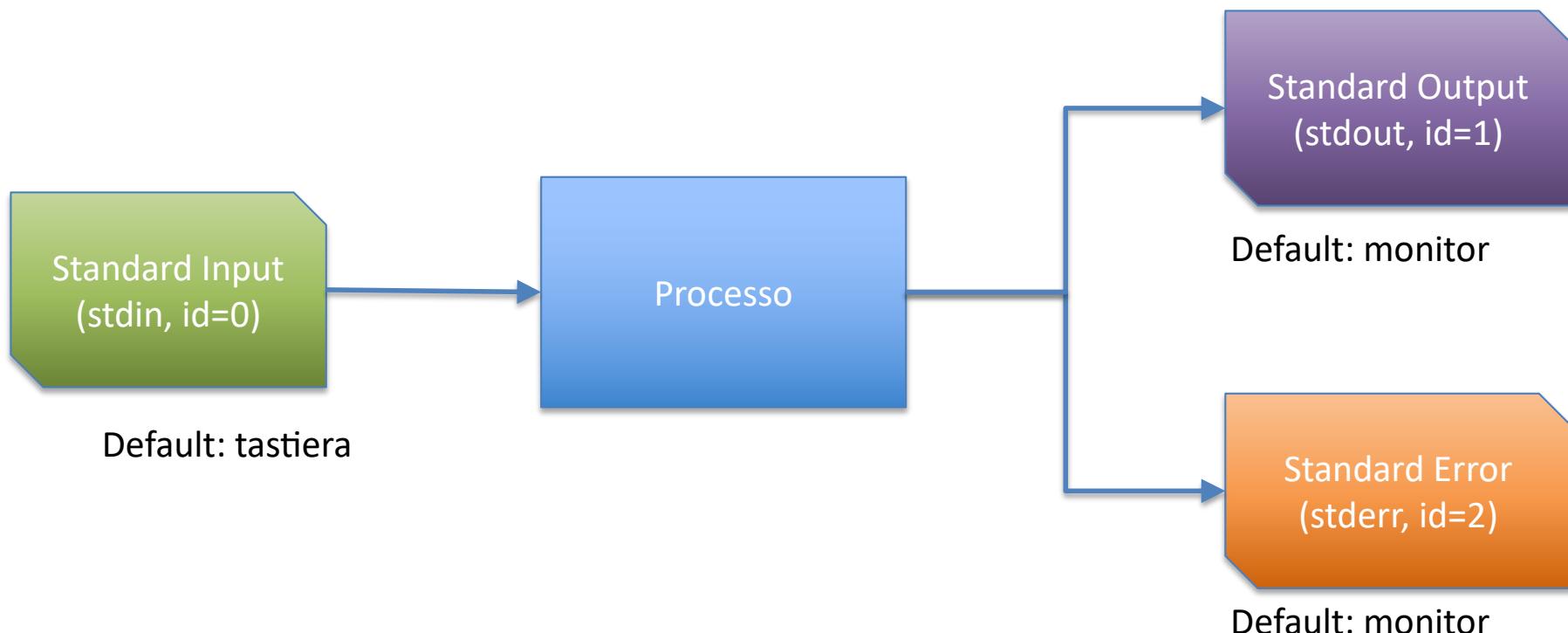
---

- Tasti freccia (su e giù) consentono di spostarsi all'interno della lista dei comandi precedenti (lo stesso elenco mostrato dal comando history)
- **ctrl-r** consente di inserire una stringa e selezionare tutti i comandi precedenti che la contengono. Ogni pressione successiva della combinazione **ctrl-r** accede agli altri comandi della stessa selezione
- **tab** auto-completa i nomi di file. Una doppia pressione (rapida) mostra l'elenco di tutte le possibilità

# Ridirezione flussi dati

---

# Flussi dati



# Filtri Unix

---

- **cat [opzione]... [file]...**
  - legge da file o stdin, scrive su stdout
- **grep [opzione]... testo [file]...**
  - Legge da file o stdin, scrive su stdout le linee che contengono <testo>
- **head [opzione]... [file]...**
  - Legge da file o stdin, scrive su stdout un subset delle righe (prime n)
- **tail [opzione]... [file]...**
  - Legge da file o stdin, scrive su stdout un subset delle righe (ultime n)
- **cut [opzione]... [file]...**
  - Legge da file o stdin, scrive su stdout un subset delle colonne del file
- **sort [opzione]... [file]...**
  - Legge da file o stdin, scrive su stdout linee ordinate
- **tee [opzione]... file**
  - Legge da stdin, sdoppia il flusso in ingresso su stdout e <file>

# Ridirezione

---

- E' possibile ridirigere input e/o output di un comando facendo sì che stdin/stdout/stderr siano sostituiti da file **in modo trasparente al comando**
- Ridirezione dell'input
  - comando < filein
- Ridirezione dell'output
  - comando > fileout (sovrascrive fileout)
  - comando >> fileout (aggiunge alla fine di fileout)

# Esempi ridirezione

---

```
$ cat /etc/passwd
```

cat legge da /etc/passwd e stampa il contenuto su stdout

```
$ cat < /etc/passwd
```

cat legge da stdin, ma il flusso proviene da /etc/passwd

```
$ sort < fin > fout
```

sort legge da stdin, ma il flusso proviene da fin

sort scrive su stdout, ma il flusso è ridiretto su fout

```
$ head fin > fout
```

head legge da fin

head scrive su stdout, ma il flusso è ridiretto su fout

## Esempi ridirezione

---

```
$ tr [:lower:] [:upper:] < fin > fout
```

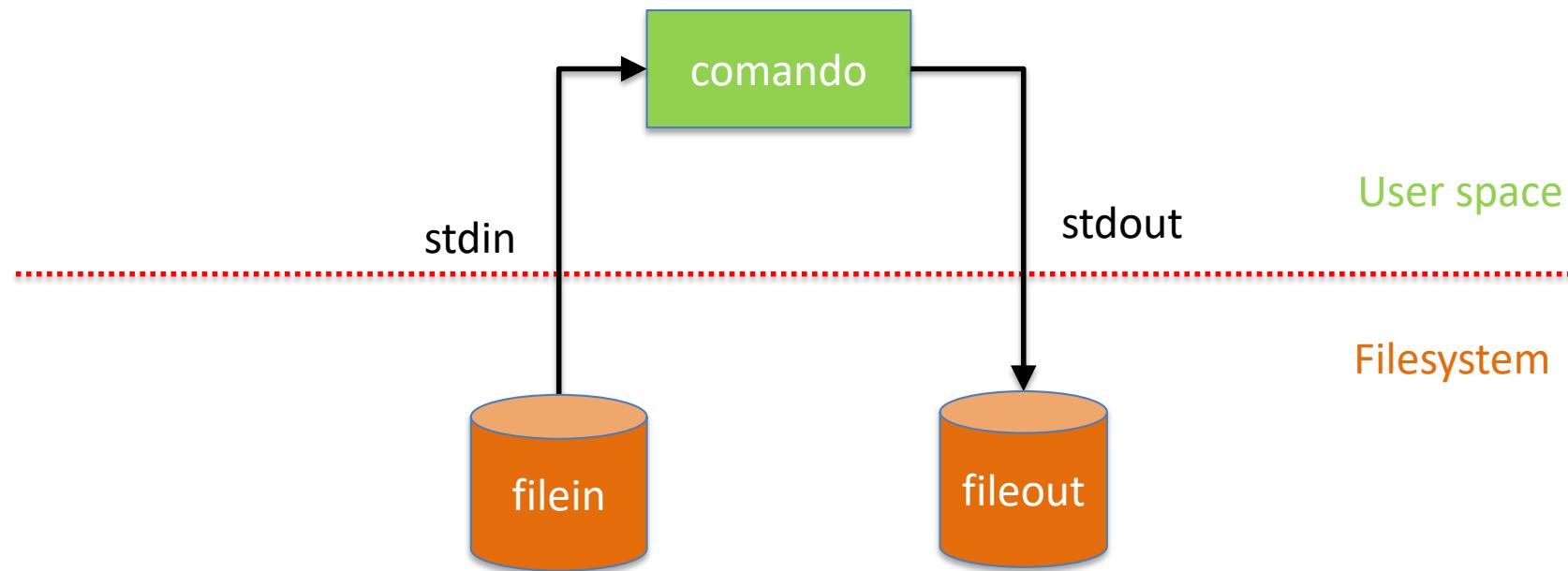
tr legge da stdin, ma il flusso proviene da fin

tr scrive su stdout, ma il flusso è ridiretto su fout

```
$ who >> users
```

who scrive su stdout, ma il flusso è ridiretto su users (append)

# Implementazione ridirezione



# Separazione stdout, stderr

---

```
$ grep nicola /etc/passwd
```

- Seleziona tutte le righe che contengono la stringa *nicola* all'interno del file */etc/passwd* e le stampa sul terminale
  - Se il file viene trovato, stampa il risultato su stdout
  - Se il file non viene trovato, stampa un errore su stderr
- E' possibile separare stdout e stderr menzionandoli in modo esplicito in forma numerica (0 = stdin, 1=stdout, 2=stderr)

```
$ grep nicola /etc/passwd 1>/dev/null
```

(scarta stdout, mostra stderr)

```
$ grep nicola /etc/passwd 2>/dev/null
```

(scarta stderr, mostra stdout)

# Separazione stdout, stderr

---

```
$ vim test.py
#!/usr/bin/env python3
import sys
sys.stdout.write("Hello stdout!\n")
sys.stderr.write("Hello stderr!\n")
sys.exit(0)
$ chmod 755 test.py
$ ./test.py 1>/dev/null
Hello stderr!
$ ./test.py 2>/dev/null
Hello stdout!
```

# /dev/null

---

- **/dev/null** File speciale (device) che scarta tutto ciò che gli viene scritto sopra. E' il buco nero di ogni sistema
- **/dev/zero** File speciale (device) che produce zeri all'infinito quando viene letto.
- **/dev/urandom** File speciale (device) che produce caratteri casuali all'infinito quando viene letto.

```
$ cat /dev/zero > fout      (premere ctrl-c)
```

```
$ cat /dev/urandom > fout (premere ctrl-c)
```

```
$ cat /dev/zero > /dev/null (zeri diretti verso il nulla cosmico)
```

## 2>&1

---

- E' possibile ridirigere un flusso all'interno di un altro flusso

```
$ grep nicola /etc/passwd 1>/dev/null 2>&1
```

```
$ grep nicola /etc/passwd >/dev/null 2>&1
```

(Stesso significato, scrittura meno chiara)

- Il flusso 2 viene ridiretto all'interno del flusso 1. Il carattere & chiarisce che non si tratta di un file di nome 1, ma del flusso numero 1 (stdout).

# Composizione comandi

---

# Combinare comandi (&&, ||)

---

- **&& AND LOGICO** Esegue il secondo comando se il primo termina con successo (ritorna 0):

```
$ true && ls
```

```
$ false && ls
```

- **|| OR LOGICO** Esegue il secondo comando se il primo fallisce (ritorna 1):

```
$ true || ls
```

```
$ false || ls
```

# Combinare comandi (;

---

- ; CONCATENAZIONE SEMPLICE Esegue il secondo cmd a prescindere dal valore di ritorno del primo

```
$ true; ls
```

```
$ false; ls
```

```
$ true; echo $?
```

```
$ false; echo $?
```

# Pipes

---

- E' sempre possibile combinare comandi utilizzando il filesystem come strumento di mediazione

```
$ sort f > f2; head -n 10 f2
```

- sort legge f, lo ordina, stampa su stdout (ridiretto su file f2). head legge le prime 10 linee di f2. Il carattere ; è utilizzato per combinare comandi sulla stessa linea.

- Approccio estremamente inefficiente. La memoria secondaria (HDD) è molto meno performante della memoria primaria (RAM). HDD SSD: ~0.1GB/S, DDR4: ~5GB/S

# Pipes

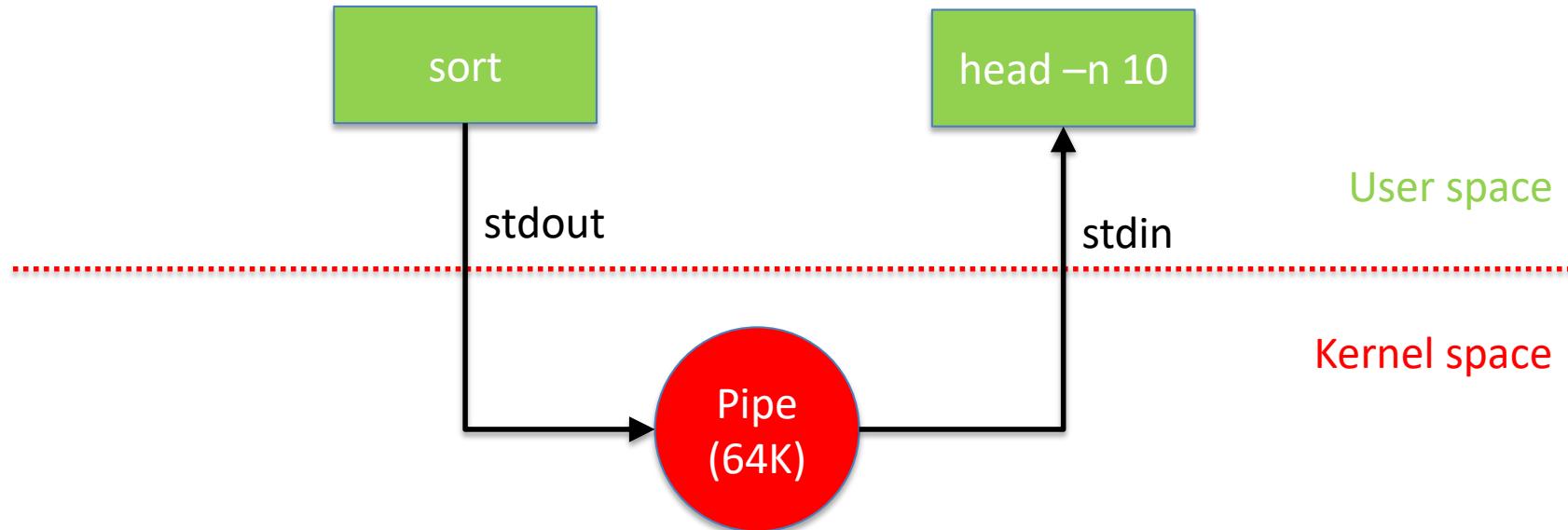
---

- L'output di un comando può esser diretto a diventare l'input di un altro comando (usando costrutto pipe '|')
- Pipe come costrutto parallelo (l'output del primo comando viene reso disponibile al secondo e consumato appena possibile, in assenza di file temporanei)

```
$ sort /etc/passwd | head  
$ sort /etc/passwd | tail -n 5  
$ sort /etc/passwd | tail -n 10 | head -n 5  
$ cat /dev/urandom | wc -c (osservare con top)
```



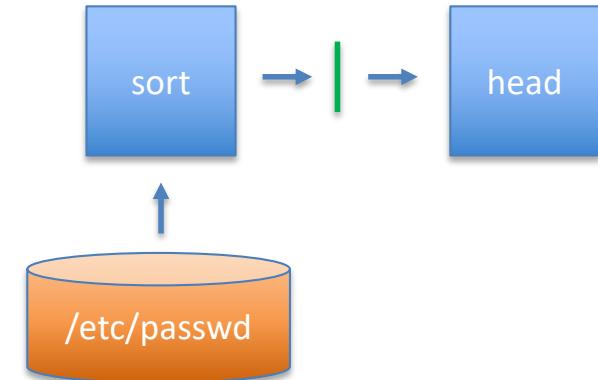
# Implementazione pipes



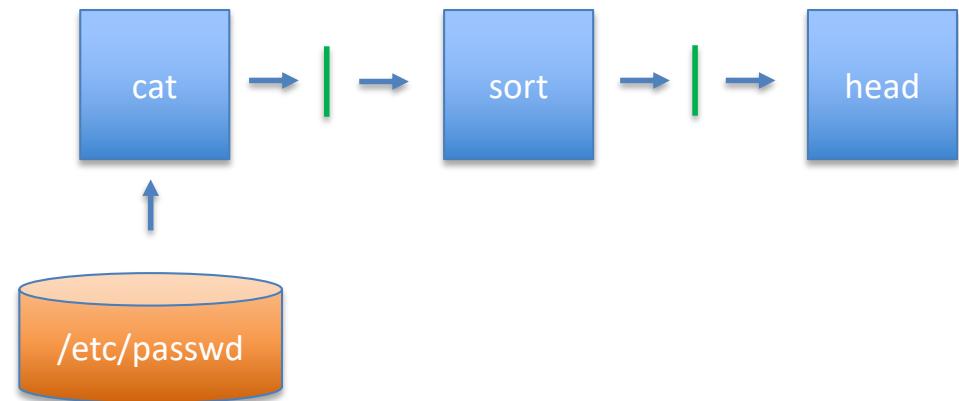
# Esempi

---

```
$ sort /etc/passwd | head
```



```
$ cat /etc/passwd | sort | head
```



# Esempi

---

```
$ who | wc -l
```

Conta gli utenti collegati al sistema

```
$ ls -l | grep ^d | sort
```

Stampa il contenuto della cartella corrente, seleziona le righe che iniziano per d (directory), ed ordina il risultato

```
$ cat /etc/passwd | cut -d ":" -f 7 | uniq -u
```

Mostra shell usate nel sistema senza ripetizioni

```
$ sort /etc/passwd | tail -n 10 | head -n 5
```

Mostra penultimi 5 utenti in ordine alfabetico

# Foreground e Background

---



&

---

- E' possibile eseguire comandi in background istruendo la shell di ritornare immediatamente un prompt con il carattere &

```
$ cat /dev/zero > /dev/null &
```

\$

- E' inoltre possibile riportare in foreground (primo piano) un processo in background con il comando **builtin fg**
- \$ **fg**

# kill

---

- E' possibile comunicare con i processi inviando loro dei segnali (man 7 signal). Il comando ortodosso per inviare segnali ai processi è kill (man kill)

```
$ ps
```

PID	TTY	TIME	CMD
19260	pts/1	00:00:00	bash
19268	pts/1	00:00:00	ps

```
$ kill -9 19260      # oppure  
$ kill -s 9 19260     # oppure  
$ kill -s SIGKILL 19260
```

# \$ man 7 signal

SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
<b>SIGINT</b>	<b>2</b>	<b>Term</b>	<b>Interrupt from keyboard</b>
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
<b>SIGKILL</b>	<b>9</b>	<b>Term</b>	<b>Kill signal</b>
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
<b>SIGSTOP</b>	<b>17,19,23</b>	<b>Stop</b>	<b>Stop process</b>
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

# ctrl-c, ctrl-z

---

- ctrl-c invia al processo in foreground il segnale SIGINT
- ctrl-z invia al processo in foreground il segnale SIGSTOP
- I processi Unix tipicamente (esistono eccezioni):
  - quando ricevono SIGINT interrompono l'esecuzione e terminano
  - quando ricevono SIGSTOP interrompono l'esecuzione ma NON terminano
- Il comando **fg** riprende l'esecuzione di un comando interrotto riportandolo in foreground
- Il comando **bg** riprende l'esecuzione di un comando interrotto mantenendolo in background

# fg, bg

---

```
$ cat /dev/zero > /dev/null
```

```
^C
```

```
$
```

```
$ cat /dev/zero > /dev/null
```

```
^Z
```

```
[1]+ Stopped      cat /dev/zero > /dev/null
```

```
$ fg
```

```
cat /dev/zero > /dev/null
```

```
$ cat /dev/zero > /dev/null
```

```
^Z
```

```
[1]+ Stopped      cat /dev/zero > /dev/null
```

```
$ bg
```

```
[1]+ cat /dev/zero > /dev/null &
```

```
$
```



# Variabili

---

# Variabili shell

---

- E' possibile definire variabili (trattate come stringhe) ed assegnare loro un valore con operatore =

```
$ VAR=3
```

- Si accede ai valori delle variabili con il carattere speciale \$

```
$ echo $VAR
```

```
$ A=1; B=nicola
```

```
$ echo $A
```

```
1
```

```
$ echo $B
```

```
nicola
```



# Visibilità variabili

---

- La visibilità delle variabili definite all'interno di una shell è limitata alla shell stessa. **Eventuali sotto-shell non ereditano le variabili.**

```
$ A=1  
$ echo $A  
1  
$ bash (sotto-shell)  
$ echo $A
```

```
$
```

# Variabili d'ambiente

---

- Per propagare variabili anche alle sotto shell si utilizzano particolari variabili chiamate d'ambiente.
- Ogni comando esegue nell'ambiente associato alla shell che lo esegue. Ogni shell eredita l'ambiente dalla shell che l'ha creata
- La prima shell ad eseguire dopo il login o dopo l'apertura di un terminale grafico, legge un file (es. `.profile/.bashrc`) che contiene fra le altre cose variabili di configurazione che vengono così propigate a tutte le shell eseguite in futuro.

## env

---

```
$ env
SHELL=/bin/bash
TERM=xterm-256color
USER=nicola
HOME=/home/nicola
LOGNAME=nicola
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

# set

---

- Il comando **builtin set** è un'alternativa al comando **env**.
- Differenze principali:
  - **set** è un comando builtin: vede variabili sia locali che d'ambiente della shell corrente
  - **env** è un comando esterno: vede, per ovvi motivi, solo variabili d'ambiente
- Il comando **builtin unset** rimuove una variabile dalla memoria

```
$ A=1  
$ unset A  
$ echo $A
```



# export

---

- E' possibile aggiungere variabili all'ambiente utilizzando il comando **export**. Le variabili esportate si comportano come variabili locali ma sono visibili anche dalle sotto-shell

```
$ export A=1
$ echo $A
1
$ bash (sotto-shell)
$ echo $A
1
$
```



# Variabile PATH

---

- I binari di sistema sono posizionati all'interno di varie directory. Le principali sono /bin, /usr/bin ma possono esisterne altre.
- La variabile d'ambiente PATH tiene traccia dell'elenco delle cartelle che contengono binari all'interno del sistema

```
$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b  
in:/sbin:/bin:/usr/games:/usr/local/games:/snap  
/bin
```

# Variabile PATH

---

- E' possibile aggiungere un nuovo percorso alla variabile PATH

```
$ export PATH="$PATH":/opt/mybin (alla fine)
```

```
$ export PATH=/opt/mybin:$PATH (all'inizio)
```

```
$ export PATH="" (svuota PATH)
```

```
$ ls
```

```
ls: No such file or directory
```

# Espansione ed inibizione

---



# Metacaratteri

---

- La shell riconosce caratteri speciali (wild cards)
  - \* una qualunque stringa di zero o più caratteri in un nome di file
  - ? un qualunque carattere in un nome di file
  - [abc] un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche range di valori: [a-g]. Per esempio ls [q-s]\* stampa tutti i file con nomi che iniziano con un carattere compreso tra q e s
  - \ segnala di non interpretare il carattere successivo come speciale

# Metacaratteri

---

\$ ls [a-p,1-7]\*[c,f,d]?

Elenca i file i cui nomi hanno come iniziale un carattere compreso tra ‘a’ e ‘p’ oppure tra ‘1’ e ‘7’, e il cui penultimo carattere sia ‘c’, ‘f’, o ‘d’

\$ ls \*\\*\*

Elenca i file che contengono, in qualunque posizione, il carattere ‘\*’

# Esecuzione in-line

---

E' possibile eseguire un comando ed utilizzarne l'output all'interno di un altro comando

```
$ echo $(pwd)  
/home/nicola  
$ echo `pwd`  
/home/nicola  
$ echo $(expr 2 + 3)  
5
```

# Espansione

---

- Comandi contenuti tra `$()` o `` (backquote) sono eseguiti e sostituiti col il risultato prodotto
- Nomi delle variabili (`$A`) sono espansi nei valori corrispondenti
- Metacaratteri `* ? [ ]` sono espansi nei nomi di file secondo un meccanismo di pattern matching

# Inibizione espansione

---

- In alcuni casi è necessario privare i caratteri speciali del loro significato, considerandoli come caratteri normali
  - \ carattere successivo è considerato come un normale carattere
  - '' (singoli apici): proteggono da qualsiasi tipo di espansione
  - ""(doppi apici) proteggono dalle espansioni con l'eccezione di
    - \$
    - \
    - ` ` (backquote)

# Inibizione espansione

---

```
$ rm *$var *
```

Rimuove i file che cominciano con \*\$var

```
$ rm *$var*
```

Rimuove i file che cominciano con \*<contenuto della variabile var>

```
$ echo $(pwd)
```

/home/nicola

```
$ echo <$(pwd)>
```

```
<$(pwd)>
```

```
$ A=1+2; B=$(expr 1 + 2)
```

In A viene memorizzata la stringa 1+2, in B la stringa 3

# Riassumendo

---

```
$ cp -r $(pwd)/ss* "$HOME"/config > /tmp/service.Log
```

- Ridirezione dell'input/output
- Esecuzione e sostituzione dei comandi \$()

$\$(\text{pwd}) \rightarrow /etc$

- Sostituzione di variabili e parametri

$\$HOME \rightarrow /home/nicola$

- Sostituzione di metacaratteri

$ss* \rightarrow /ssh/$