

Università di Ferrara
Laurea Triennale in Informatica
A.A. 2022-2023
Sistemi Operativi e Laboratorio

**2. Introduzione a UNIX shell
e file comandi**

Prof. Carlo Giannelli

`http://www.unife.it/scienze/informatica/insegnamenti/
sistemi-operativi-laboratorio`

`http://docente.unife.it/carlo.giannelli`

`https://ds.unife.it/people/carlo.giannelli`

Shell

Programma che permette di far **interagire l'utente (interfaccia testuale) con SO tramite comandi**

- ❑ resta in attesa di un comando...
- ❑ ... mandandolo in esecuzione alla pressione di <ENTER>

In realtà (lo vedremo ampiamente) **shell è un interprete comandi evoluto**

- ❑ potente **linguaggio di scripting**
- ❑ interpreta ed esegue comandi da **standard input** o da **file comandi**

Differenti shell

- La shell non è unica, un sistema può metterne a disposizione varie
 - ❑ **Bourne shell** (standard), C shell, Korn shell, ...
 - ❑ L'implementazione della **bourne shell in Linux** è **bash** (`/bin/bash`)
- Ogni utente può indicare la shell preferita
 - ❑ La scelta viene memorizzata in `/etc/passwd`, un file contenente le informazioni di tutti gli utenti del sistema
- La shell di login è quella che richiede inizialmente i dati di accesso all'utente
 - ❑ Per **ogni utente connesso** viene generato un **processo dedicato** (che esegue la shell)

Ciclo di esecuzione della shell

```
loop forever
  <LOGIN>
  do
    <ricevi comando da file di input>
    <interpreta comando>
    <esegui comando>
    while (! <EOF>)
  <LOGOUT>
end loop
```

Accesso al sistema: login

Per accedere al sistema bisogna possedere una coppia **username e password**

❑ NOTA: UNIX è case-sensitive

SO verifica le credenziali dell'utente e manda in esecuzione la sua **shell di preferenza**, posizionandolo in un **direttorio di partenza**

❑ Entrambe le informazioni si trovano in `/etc/passwd`

Comando `passwd`

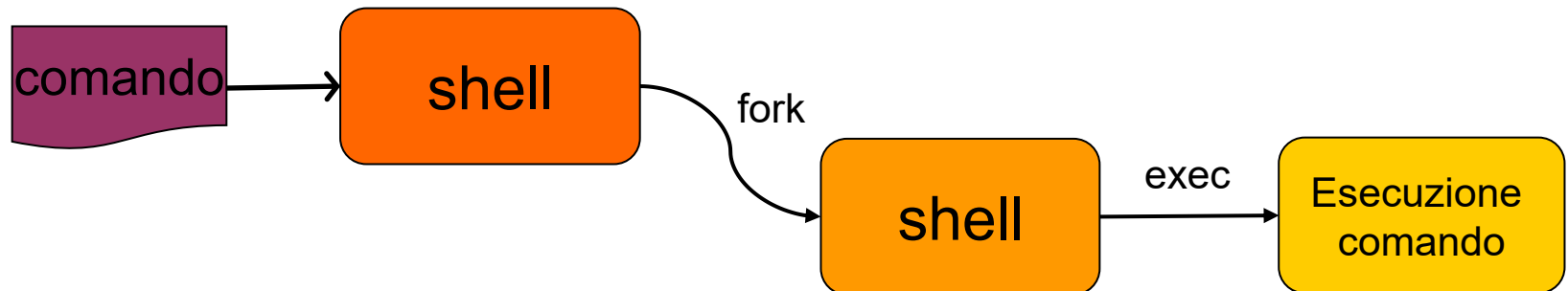
- È possibile **cambiare la propria password** di utente, mediante il comando `passwd`
- Se ci si dimentica della password, bisogna chiedere all'amministratore di sistema (utente root)

Uscita dal sistema: logout

- Per uscire da una shell qualsiasi si può utilizzare il comando **exit** (che invoca la system call **exit()** per quel processo)
- Per uscire dalla shell di login
 - ❑ **logout**
 - ❑ **CTRL+D** (che corrisponde al carattere <EOF>)
 - ❑ **CTRL+C**
- Per rientrare nel sistema bisogna effettuare un nuovo login

Esecuzione di un comando

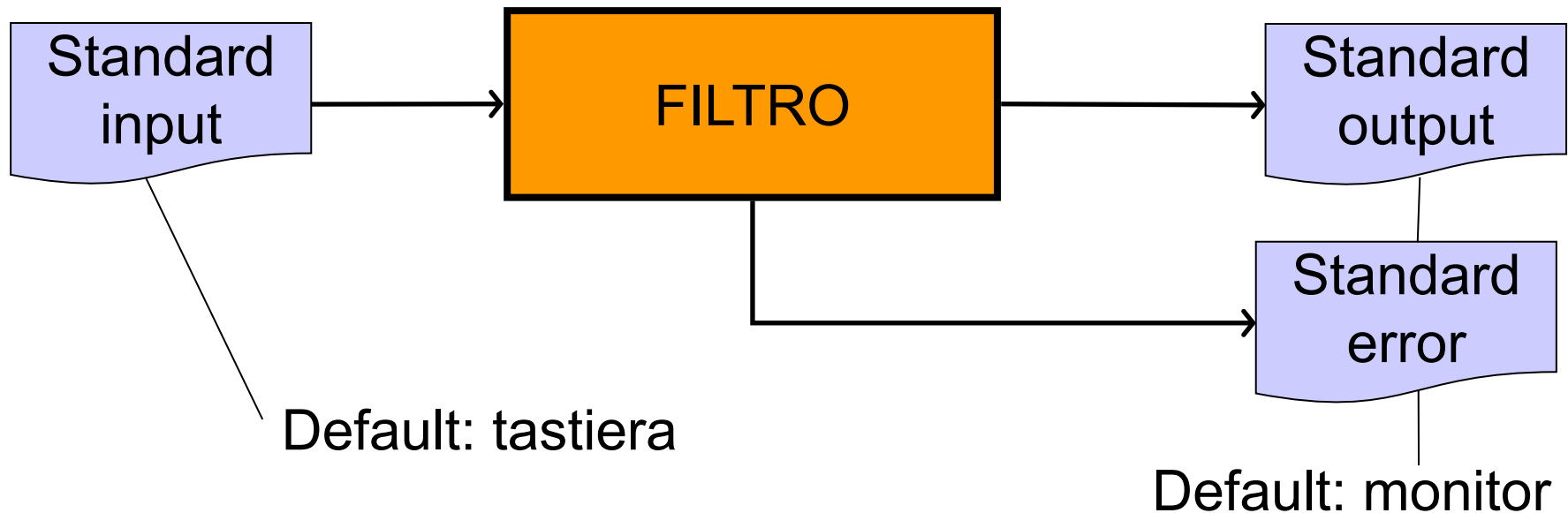
- Ogni comando richiede al SO l'esecuzione di una particolare azione
- I **comandi principali** del sistema si trovano nella directory **/bin**
- Possibilità di **realizzare nuovi comandi (scripting)**
- Per ogni comando, shell **genera un processo figlio dedicato alla sua esecuzione**
 - ❑ Il processo padre **attende la terminazione del comando** (foreground) o **prosegue in parallelo** (background)



Comandi e input/output

❑ I comandi UNIX si comportano come FILTRI

- ❑ un filtro è un programma che riceve un ingresso da un input e produce il risultato su uno o più output



Manuale

Esiste un **manuale on-line** (**man**), consultabile per informazioni su ogni comando Linux. Indica:

- ❑ **formato del comando (input) e risultato atteso (output)**
- ❑ **descrizione delle opzioni**
- ❑ possibili restrizioni
- ❑ file di sistema interessati dal comando
- ❑ comandi correlati
- ❑ eventuali bug

Per uscire dal manuale, digitare :q (quit per editor tipo vi)

Formato dei comandi

Tipicamente: ***nome –opzioni argomenti***

Esempio: **`ls -l temp.txt`**

Convenzione nella rappresentazione della sintassi comandi:

- ❑ se un'opzione o un argomento possono essere omessi, si indicano tra quadre **[opzione]**
- ❑ se due opzioni/argomenti sono mutuamente esclusivi, vengono separati da | **arg1 | arg2**
- ❑ quando un arg può essere ripetuto n volte, si aggiungono dei puntini **arg...**

Cenni pratici introduttivi all'utilizzo del file system Linux

File

File come **risorsa logica** costituita da **sequenza di bit**, a cui viene dato un nome

Astrazione molto potente che consente di **trattare allo stesso modo entità fisicamente diverse** come file di testo, dischi rigidi, stampanti, direttori, tastiera, video, ...

- **Ordinari**
 - ❑ archivi di dati, comandi, programmi sorgente, eseguibili, ...
- **Directory**
 - ❑ gestiti direttamente solo da SO, contengono riferimenti a file
- **Speciali**
 - ❑ dispositivi hardware, memoria centrale, hard disk, ...

In aggiunta, anche:

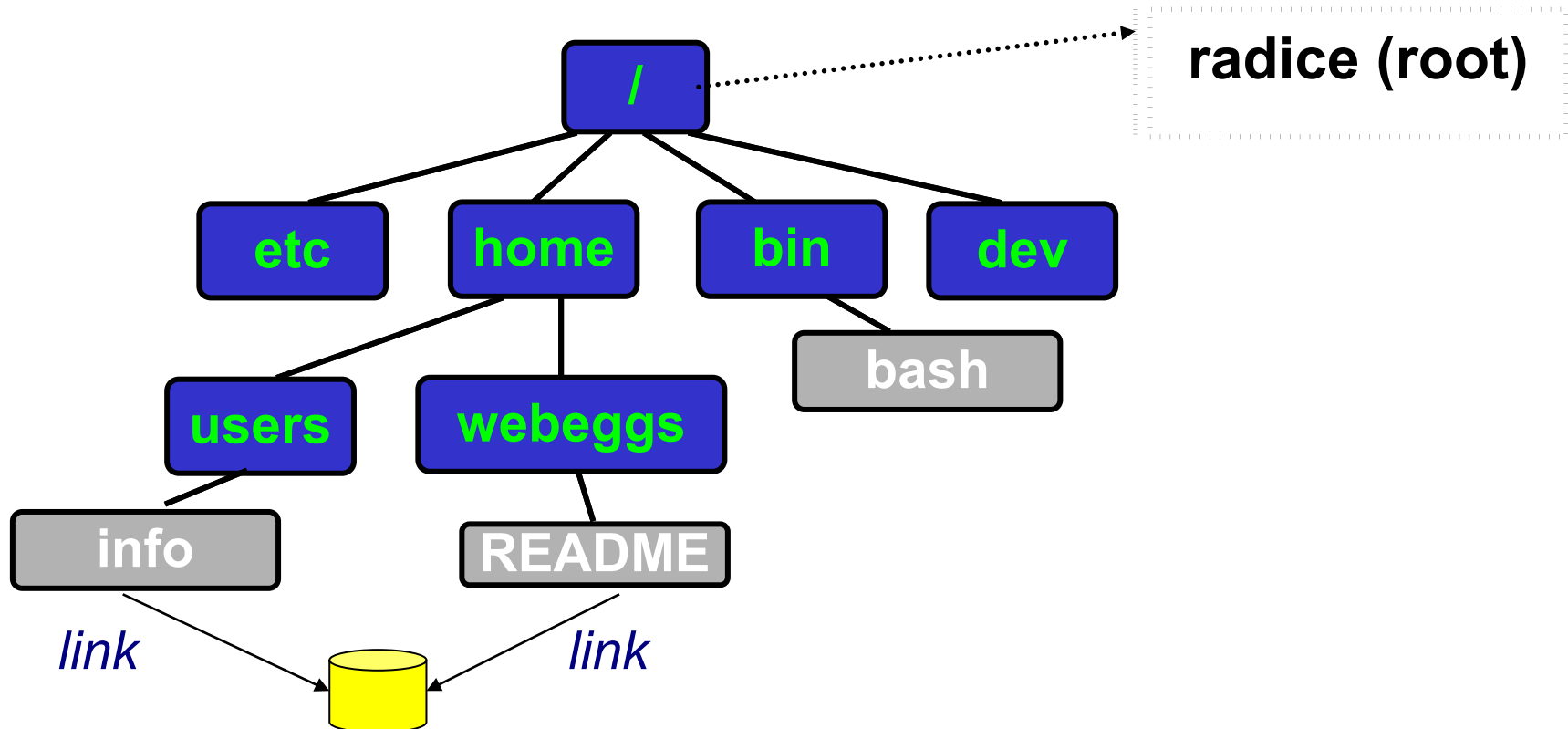
- FIFO (pipe) - file per la comunicazione tra processi
- soft link - riferimenti (puntatori) ad altri file o direttori

File: nomi

- È possibile nominare un file con una **qualsiasi sequenza di caratteri (max 255)**, a eccezione di '.' e '..'
- È sconsigliabile utilizzare per il nome di file dei caratteri speciali, ad es. **metacaratteri e segni di punteggiatura**
- Ad ogni file possono essere associati **uno o più nomi simbolici (link)** ma ad ogni file è associato **uno e un solo descrittore (i-node)** identificato da un intero (i-number)

Directory

File system Linux è organizzato come un grafo diretto aciclico (DAG)



Gerarchie di directory

- All'atto del login, l'utente può cominciare a operare all'interno di una specifica directory (**home**). In seguito è possibile cambiare directory
- È possibile visualizzare il percorso completo attraverso il **comando** `pwd` (print working directory)
- Essendo i file organizzati in **gerarchie di directory**, SO mette a disposizione dei comandi per muoversi all'interno di essi

Nomi relativi/assoluti

Ogni utente può specificare un file attraverso

- ❑ **nome relativo**: è riferito alla posizione dell'utente nel file system (direttorio corrente)
- ❑ **nome assoluto**: è riferito alla radice della gerarchia /

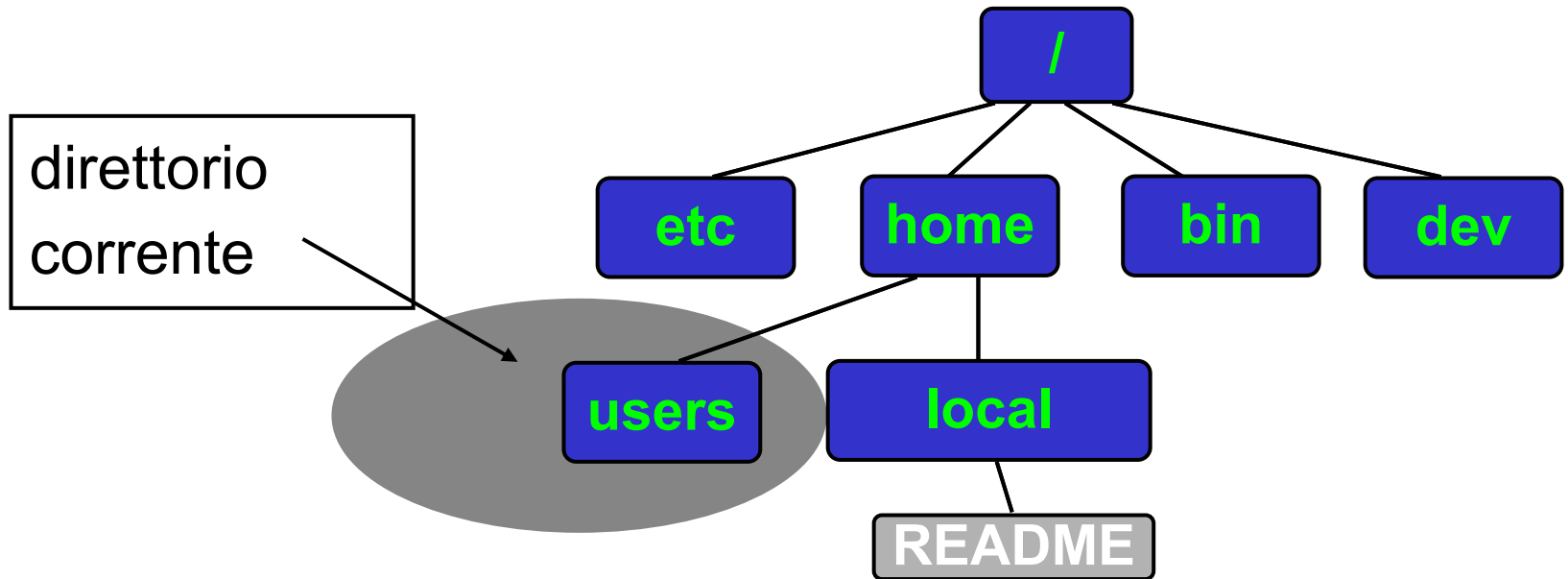
Nomi particolari

- ❑ **.** è il direttorio corrente (visualizzato da **pwd**)
- ❑ **..** è il direttorio 'padre'
- ❑ **~** è la propria home utente

Il comando **cd** **permette di spostarsi all'interno del file system**, utilizzando sia nomi relativi che assoluti

- ❑ **cd** senza parametri porta alla home dell'utente

Nomi relativi/assoluti: esempio



nome assoluto: `/home/local/README`

nome relativo: `../local/README`

Link

Le informazioni contenute in uno **stesso file** possono essere **visibili come file diversi**, tramite “riferimenti” (link) allo stesso file fisico

- SO considera e gestisce la molteplicità possibile di riferimenti:
 - ❑ se un file viene cancellato, le **informazioni sono veramente eliminate solo se non ci sono altri link a esso**
 - ❑ Il link **cambia i diritti?** → **Meglio di no**

Due tipi di link:

- **link fisici** (si collegano le strutture del file system)
- **link simbolici** (si collegano solo i nomi)

comando: **ln [-s]**

Gestione file: comando **ls**

Consente di **visualizzare nomi di file**

- varie opzioni: esempio **ls -l** per avere più informazioni (non solo il nome del file)
- possibilità di usare **metacaratteri (wildcard)**
 - ❑ Per es. se esistono i file **f1, f2, f3, f4**
 - ci si può riferire a essi scrivendo: **f***
 - o più precisamente **f[1-4]**

Più avanti studieremo meglio i metacaratteri e le modalità con cui vengono gestiti esattamente dalla shell

Opzioni del comando ls...

`ls [-opzioni...] [file...]`

Alcune opzioni

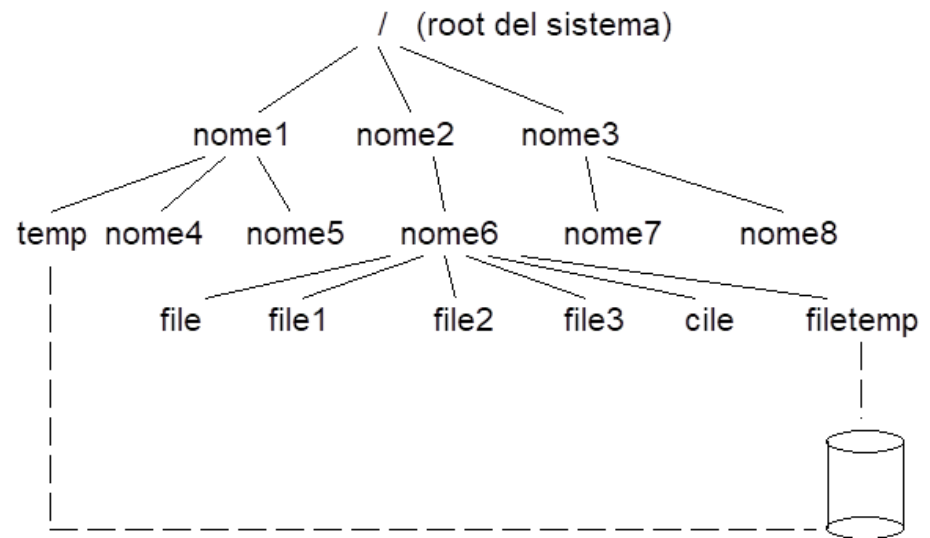
- ❑ **l** (long format): per ogni file una linea che contiene **diritti**, **numero di link**, **proprietario** del file, **gruppo** del proprietario, **occupazione di disco** (blocchi), **data e ora** dell'ultima modifica o dell'ultimo accesso e **nome**
- ❑ **t** (time): la lista è **ordinata per data** dell'ultima modifica
- ❑ **u** la lista è ordinata per data dell'ultimo accesso
- ❑ **r** (reverse order): inverte l'ordine
- ❑ **a** (all files): fornisce una **lista completa** (normalmente i file il cui nome comincia con il punto non vengono visualizzati)
- ❑ **F** (classify): indica anche il tipo di file (eseguibile: *, directory: /, link simbolico: @, FIFO: |, socket: =, niente per file regolari)

Comandi vari di gestione

- **Creazione/gestione di directory**
 - ❑ **mkdir** <nomedir> *creazione di un nuovo direttorio*
 - ❑ **rmdir** <nomedir> *cancellazione di un direttorio*
 - ❑ **cd** <nomedir> *cambio di direttorio*
 - ❑ **pwd** *stampa il direttorio corrente*
 - ❑ **ls** [<nomedir>] *visualizz. contenuto del direttorio*
- **Trattamento file**
 - ❑ **ln** <vecchionome> <nuovonome> *link*
 - ❑ **cp** <filesorgente> <filedestinazione> *copia*
 - ❑ **mv** <vecchionome> <nuovonome> *rinom. / spost.*
 - ❑ **rm** <nomefile> *cancellazione*
 - ❑ **cat** <nomefile> *visualizzazione*

Comando ln, link

- Le informazioni contenute in uno stesso file possono essere visibili come file diversi, tramite “riferimenti” (link) allo stesso file fisico.
- SO considera e tratta il tutto: se un file viene cancellato, le informazioni sono veramente eliminate solo se non ci sono altri link a esso



comando: ln [-s]

- ln /nome2/nome6/filetemp /nome1/temp

cat

Il comando cat stampa a video il contenuto di un file:

```
cgiannelli@linuxdid:~$ cat /etc/passwd
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
...
```

`/etc/passwd` is a text file that contains the attributes of (i.e., basic information about) each user or account on a computer running Linux or another Unix-like operating system

more (e less)

- Il comando **cat** effettua una semplicissima stampa del contenuto a video di un file. Per una visualizzazione a pagine, certamente più comoda per file di larghe dimensioni, è necessario utilizzare un comando di paginazione.
- Il comando **more** permette di visualizzare un file una pagina per volta.
- Esistono diverse alternative più recenti e potenti al comando **more**, tra cui va sicuramente segnalato il comando **less** (che permette anche di far scorrere la visualizzazione all'indietro).

echo

- Il comando **echo** stampa a video la stringa passatagli come parametro:

```
cgiannelli@linuxdid:~$ echo Hello world!  
Hello world!
```

- Il comando `echo` rappresenta la primitiva fondamentale che la shell mette a disposizione per stampare informazioni a video.

sort

- Il comando **sort** riordina le righe di un file in ordine alfabetico

- Esempi e opzioni:

```
sort nomefile.txt      # elenca le righe del file nomefile.txt
                        # in ordine alfabetico crescente
sort -r nomefile.txt   # elenca le righe del file nomefile.txt
                        # in ordine alfabetico decrescente
```

- Il comando sort ha molte opzioni:
 - ❑ -o <nomefileout> stampa su file
 - ❑ -n interpreta le righe del file come numeri
 - ❑ -k <n> ordina il file secondo il contenuto della n-esima colonna

diff

- Il comando **diff** stampa la differenza tra il contenuto di due file
- Esempio:

```
diff <file1> <file2>      # mostra (solo) le righe diverse
```
- Il comando diff è molto utilizzato nella gestione del codice sorgente

WC

- Il comando **wc** stampa il numero di righe, parole, o caratteri contenuti in un file

- Esempi e opzioni:

```
wc -l nomefile.txt    # conta le linee (opzione l)  
                      # contenute nel file nomefile.txt  
  
wc -w nomefile.txt    # conta le parole (opzione w)  
                      # contenute nel file nomefile.txt  
  
wc -c nomefile.txt    # conta i caratteri (opzione c)  
                      # contenuti nel file nomefile.txt
```

grep

- Il comando **grep** seleziona le righe di un file che contengono la stringa passata come parametro e le stampa a video

- Esempi:

```
grep stringa nomefile.txt    # seleziona le righe del file  
                             # nomefile.txt che contengono il testo "stringa"  
                             # e le stampa a video
```

```
grep -c stringa nomefile.txt    # conta le righe del file  
                             # nomefile.txt che contengono il testo "stringa"  
                             # e stampa il numero a video
```

grep

grep -r stringa dir # seleziona le righe di tutti i file
nella directory dir che contengono il
testo “stringa” e le stampa a video,
ricorsivo sulle sotto directory

- Il comando grep ha numerosissime opzioni ed è utilizzatissimo, specialmente nella gestione e manipolazione di file di configurazione e di codice sorgente

head

- Il comando **head** mostra le prime righe di un file

- Esempi e opzioni:

<code>head -n 15 nomefile.txt</code>	<code># mostra le prime 15 righe del file</code>
<code>head -c 30 nomefile.txt</code>	<code># mostra i primi 30 caratteri del file</code>

tail

- Il comando **tail** mostra le ultime righe di un file

- Esempi e opzioni:

`tail -n 15 nomefile.txt` # mostra le ultime 15 righe del file

`tail -c 30 nomefile.txt` # mostra gli ultimi 30 caratteri del file

time

- Il comando **time** cronometra il tempo di esecuzione di un comando
- Esempi:
- `cgiannelli@linuxdid:~$ time ls foto.jpeg`
foto.jpeg
ls foto.jpeg 0,00s user 0,00s system 0% cpu 0,002 total

who

- Il comando **who** mostra gli utenti attualmente collegati al sistema (ovverosia che hanno eseguito il login sulla macchina)

- Esempio:

```
cgiannelli@linuxdid:~$ who
```

```
cgiannelli :0      2013-12-15 11:01
```

```
cgiannelli pts/0    2013-12-15 11:02 (:0)
```

```
cgiannelli pts/1    2013-12-15 12:14 (:0)
```

```
cgiannelli pts/3    2013-12-16 18:19 (:0)
```

man

- In ambiente Unix, il comando **man** rappresenta l'help di sistema
- Esempio:
man grep # mostra l'help (guida) del comando grep
- Oltre al comando man, le moderne distribuzioni di Linux mettono a disposizione anche il comando **info**

ps

Un processo utente in genere viene attivato a partire da un comando (da cui prende il nome). Ad es., dopo aver mandato in esecuzione il comando `hw`, verrà visualizzato un processo dal nome `hw`.

Tramite `ps` si può vedere la lista dei processi attivi

```
cgianelli@info1-linux:~$ ps
```

PID	TTY	STAT	TIME	COMMAND
4837	p2	S	0:00	-bash
6945	p2	S	0:00	sleep 5s
6948	p2	R	0:00	ps

Comando `ps` molto utile quando si lancia l'esecuzione di programmi di sistema con errori di programmazione (guardare su `man` le varie opzioni `ps`)

ps

cgiannelli@linuxdid:~\$ ps au

```
user@YOGA-CG-UNIFE:~$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         6  0.0  0.0   8900    216 tty1      Ss   11:40    0:00 /init ro
user        7  0.0  0.0  20076   6708 tty1      S    11:40    0:00 -bash
user       92  0.0  0.0  17380   1920 tty1      R    11:59    0:00 ps au
user@YOGA-CG-UNIFE:~$
```

Cosa è il PID?

ps

Alcune opzioni importanti:

- a Mostra anche i processi degli altri utenti
- u Fornisce nome dell'utente che ha lanciato il processo e l'ora in cui il processo è stato lanciato
- x Mostra anche i processi senza terminale di controllo (processi demoni)

Attenzione: si ricordi di non usare il trattino (-) per specificare le opzioni del comando ps, in quanto ps adotta la sintassi di tipo “extended BSD” che non prevede l'uso di trattini.

Se si vuole un aggiornamento periodico dello stato dei processi correnti, si usi il comando **top**.

top

- Il comando **top** fornisce una visione dinamica in real-time del sistema corrente. Esso visualizza continuamente informazioni sull'utilizzo del sistema (memoria fisica e virtuale, CPU, ecc.) e sui processi che usano la maggiore share di CPU.
- Esiste anche una versione più avanzata e moderna di top, chiamata **htop**. Al contrario di top, di solito htop non è incluso tra le utility di base disponibili sulle macchine Linux e va esplicitamente installato.

pgrep e pkill

Il comando **pgrep** restituisce i pid (process id) dei processi che corrispondono alle caratteristiche richieste (nome processo, utente, ...).

Ad esempio, per ottenere il pid del processo Skype:

```
cgiannelli@linuxdid:~$ pgrep skype  
9280
```

Per ottenere il pid del processo demone sshd appartenente all'utente root:

```
cgiannelli@linuxdid:~$ pgrep -u root sshd  
3488
```

pkill presenta la stessa interfaccia comandi di pgrep, ma anziché stampare a video il pid dei processi, li uccide (in realtà invia loro un segnale SIGTERM).

Terminazione forzata di un processo

È possibile 'terminare forzatamente' un processo tramite il comando `kill`

Ad esempio:

- `kill -9 <PID>` provoca l'invio di un segnale **SIGKILL** (forza la terminazione del processo che lo riceve e non può essere ignorato) **al processo identificato da PID**

❑ Esempio: `kill -9 6944`

Per conoscere il PID di un determinato processo, si può utilizzare il comando `ps` oppure il comando `pgrep`

Segnali e interruzioni

È possibile **interrompere un processo** (purché se ne abbiano i diritti...) `kill -s <PID>`

provoca **l'invio di un segnale** (individuato dal parametro `s`) al processo identificato dal PID - vedremo ampiamente il concetto di segnale più avanti nel corso...

`kill -9` è solo un esempio: **9 corrisponde a SIGKILL**, che provoca la **terminazione incondizionata del processo (segnale non mascherabile) e dei figli (ricorsivamente)**

Alcuni tra i segnali più comuni:

- ▣ **CTRL-C** (invia un **SIGINT**, terminazione del processo attualmente in foreground, `kill -2`)
- ▣ **CTRL-Z** (invia un **SIGTSTP**, sospensione di un processo, `kill -20`)

`kill -l` fornisce la lista dei segnali

Utenti e gruppi

Utenti e gruppi

- **Sistema multiutente** \Rightarrow problemi di privacy e di possibili interferenze: necessità di **proteggere/nascondere informazione**
- Concetto di gruppo (es. staff, utenti, studenti, ...): possibilità di lavorare sugli stessi documenti
- **Ogni utente appartiene a un gruppo** ma può far parte anche di altri a seconda delle esigenze e configurazioni
- Comandi relativi all'identità dell'utente
 - ❑ **whoami**
 - ❑ **id**

Informazioni legate ai file

```
host133-63:~ carlo$ ls -l
```

```
total 8
```

tot. spazio occupato (blocchi)

```
drwx----- 3 paolo prof 102 May 18 22:49 Desktop
drwx----- 3 paolo prof 102 May 18 22:49 Documents
-rw-r--r-- 1 pippo stud 29 May 19 00:10 f1.txt
-rw-r--r-- 1 carlo prof 0 May 18 22:53 f2
```

nome file

data ultima modifica

dimensione (byte)

gruppo

proprietario

numero di (hard) link

permessi

tipo di file

Protezione dei file

- Molti utenti
 - ❑ Necessità di **regolare gli accessi** alle informazioni
- Per un file, esistono 3 tipi di utilizzatori:
 - ❑ proprietario, **user**
 - ❑ gruppo del proprietario, **group**
 - ❑ tutti gli altri utenti, **others**
- Per ogni tipo di utilizzatore, si distinguono tre modi di accesso al file:
 - ❑ **lettura (r)**
 - ❑ **scrittura (w)**
 - ❑ **esecuzione (x)** (per una directory significa list del contenuto)
- Ogni file è marcato con
 - ❑ **User-ID e Group-ID del proprietario**
 - ❑ **12 bit di protezione**

Bit di protezione

12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	1	1	1	1	0	0	1	0	0
SUID	SGID	Sticky	R	W	X	R	W	X	R	W	X
			User			Group			Others		
			PERMESSI								

Sticky bit: il sistema cerca di **mantenere in memoria l'immagine del programma**, anche se non è in esecuzione

SUID e SGID

- SUID (Set User ID) (identificatore di utente effettivo)
 - ❑ Si applica a un file di **programma eseguibile solamente**
 - ❑ **Se vale 1**, fa sì che **l'utente** che sta eseguendo quel programma **venga considerato il proprietario di quel file (solo per la durata della esecuzione)**
- È necessario per consentire operazioni di **lettura/scrittura su file di sistema**, che l'utente non avrebbe il diritto di leggere/modificare.
 - ❑ Esempio: **mkdir** crea un direttorio, ma per farlo deve anche **modificare alcune aree di sistema** (file di proprietà di root), che non potrebbero essere modificate da un utente. Solo SUID lo rende possibile
- SGID bit: come SUID bit, per il gruppo

Protezione e diritti su file

Per variare i bit di protezione:

- ❑ `chmod [u g o] [+ -] [rwx] <nomefile>`

I permessi possono essere concessi o negati dal solo **proprietario del file**

Esempi di variazione dei bit di protezione:

- ❑ `chmod 0755 /usr/dir/file`

0	0	0	1	1	1	1	0	1	1	0	1
SUID	SGID	Sticky	R	W	X	R	W	X	R	W	X
			User			Group			Others		

- ❑ `chmod u-w fileimportante`

Altri comandi:

- ❑ `chown <nomeutente> <nomefile>`

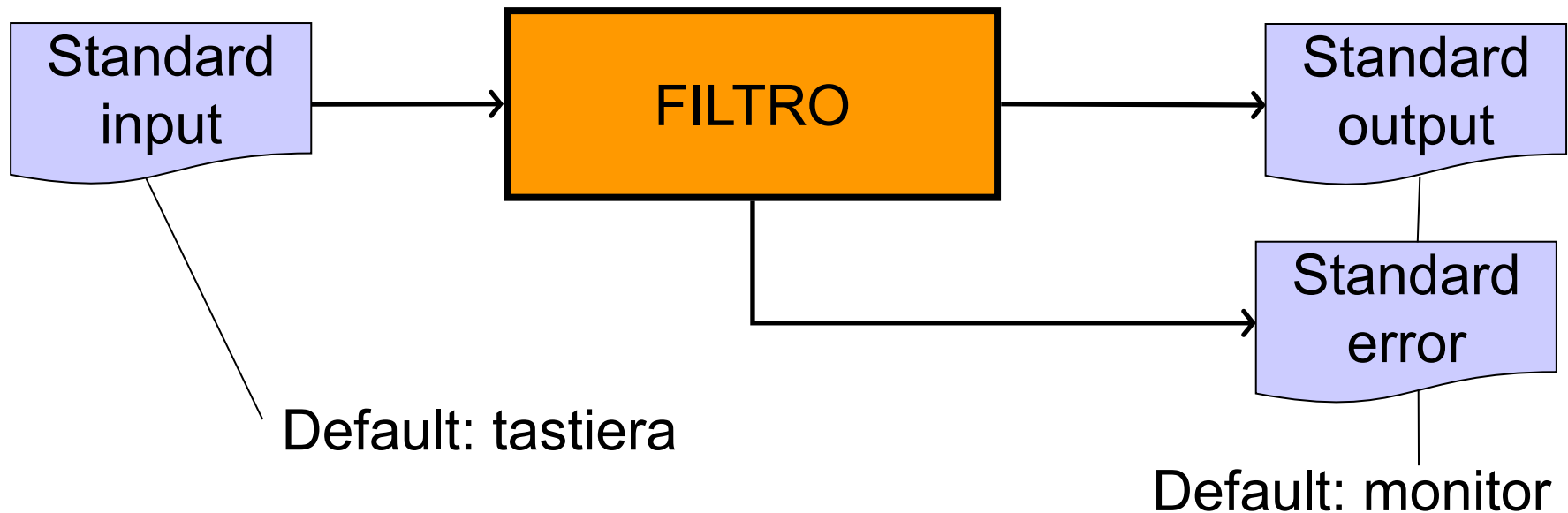
- ❑ `chgrp <nomegruppo> <nomefile>`

Comandi, piping e ridirezione

Comandi e input/output

I comandi **UNIX** si comportano come **FILTRI**

- ❑ filtro è un programma che riceve il suo **ingresso da standard input** e produce il **risultato su standard output** (trasformazione di dati)



Comandi shell Linux: filtri

Alcuni esempi:

- ❑ **grep** <testo> [<file>...]

Ricerca di testo. Input: (lista di) file. Output: video

- ❑ **tee** <file>

Scrive l'input sia su file, sia sul canale di output

- ❑ **sort** [<file>...]

Ordina alfabeticamente le linee. Input: (lista di) file.
Output: video

- ❑ **rev** <file>




Inverte l'ordine delle linee di file. Output: video

- ❑ **cut** [-options] <file>

Seleziona colonne da file. Output: video

Ridirezione di input e output

Possibile ridirigere input e/o output di un comando facendo sì che non si legga da stdin (e/o non si scriva su stdout) **ma da file**

- ❑ **senza cambiare il comando**
- ❑ **completa omogeneità tra dispositivi e file**
- Ridirezione dell'input
 - ❑ **comando < file_input** 
- Ridirezione dell'output
 - ❑ **comando > file_output** 
 - ❑ **comando >> file_output** 

Esempi

- `ls -l > file`

File conterrà il risultato di `ls -l`

- `sort < file > file2`

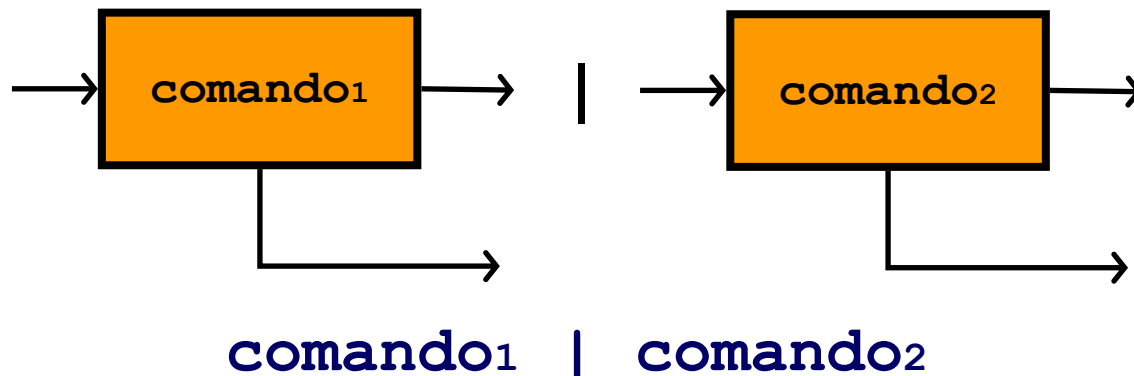
Ordina il contenuto di `file` scrivendo il risultato su `file2`

- Cosa succede con `> file` ?

Piping

L'output di un comando può esser diretto a diventare l'input di un altro comando (piping)

- ❑ In DOS: **realizzazione con file temporanei** (primo comando scrive sul file temporaneo, secondo legge da questo)
- ❑ In UNIX: **pipe come costruito parallelo** (l'output del primo comando viene reso disponibile al secondo e consumato appena possibile, non ci sono file temporanei)
- Si realizza con il carattere speciale '|'



Esempi di piping

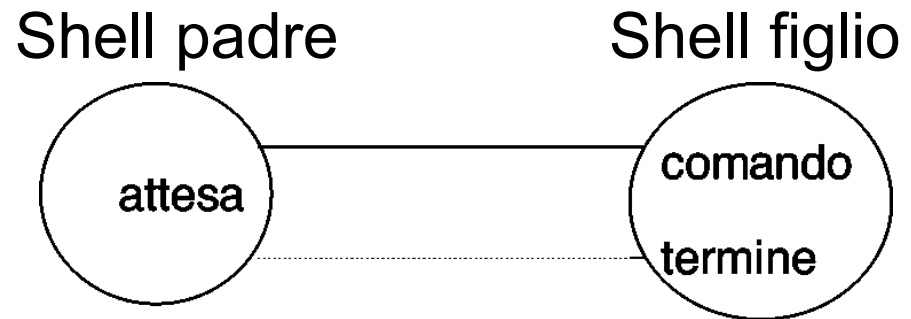
- `who | wc -l`
 - ▣ Conta gli utenti collegati
- `ls -l | grep ^d | rev | cut -d' ' -f1 | rev`
 - ▣ Che cosa fa? Semplicemente mostra i nomi dei sottodirettori della directory corrente
 - `ls -l` lista i file del direttorio corrente
 - `grep` filtra le righe che cominciano con la lettera d (pattern `^d`, vedere il `man`)
 - ovvero le directory (il primo carattere rappresenta il tipo di file)
 - `rev` rovescia l'output di `grep`
 - `cut` taglia la prima colonna dell'output passato da `rev`, considerando lo spazio come delimitatore (vedi `man`)
 - quindi, poiché `rev` ha rovesciato righe prodotte da `ls -l`, estrae il nome dei direttori 'al contrario'
 - `rev` raddrizza i nomi dei direttori

Suggerimento: aggiungere i comandi uno alla volta (per vedere cosa viene prodotto in output da ogni pezzo della pipe)

Esecuzione di comandi in Shell

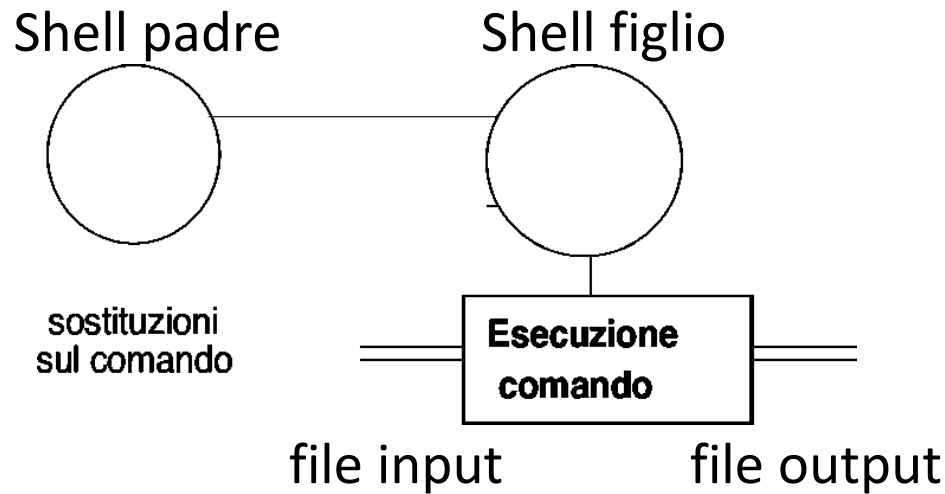
- In UNIX alcuni comandi sono eseguiti direttamente dalla shell corrente (comandi built-in)
- Invece **molti comandi sono eseguiti da una nuova shell** creata al momento

- La shell attiva mette in esecuzione una seconda shell che:
 - ❑ esegue le sostituzioni dei metacaratteri e dei parametri (parsing)
 - ❑ cerca il comando
 - ❑ esegue il comando



- La shell padre attende il completamento dell'esecuzione della sotto-shell (comportamento sincrono)

Esecuzione di un comando



Il comando esegue avendo collegato

- il proprio input al file di input
 - il proprio output al file di output
- specificati dalla shell di lancio

Schema di un processore di comandi

procedure shell (ambiente, filecomandi);

< eredita ambiente (esportato) del padre, via copia>

begin

repeat

<leggi comando da filecomandi>

if < è **comando built-in**> then

<modifica direttamente ambiente>;

else if <è **comando eseguibile**> then

<esecuzione del comando via nuova shell>

else if <è **nuovo filecomandi**> then

shell (ambiente, nuovofilecomandi);

else < errore>;

endif

until <fine file>

end shell;

Per abortire il comando corrente: CTRL-C

Non creare un file
comandi ricorsivo
senza una condizione
di terminazione !!

Rientranza della shell

- Una shell è un programma che esegue i comandi, forniti da terminale o da file
- Si invocano le shell come i normali comandi eseguibili con il loro nome

sh [<filecomandi>]

csh [<filecomandi>]

- **Le invocazioni attivano un processo che esegue la shell**
- Le shell sono **RIENTRANTI**: più processi possono condividere il codice senza errori e interferenze

sh

sh

csh

ps # quanti processi si vedono?

Metacaratteri ed espansione

Metacaratteri

Shell riconosce **caratteri speciali (wild card)**

***** una qualunque stringa di zero o più caratteri in un nome di file

? un qualunque carattere in un nome di file

[zfc] un qualunque carattere, in un nome di file, compreso tra quelli nell'insieme. Anche **range** di valori: **[a-d]**. Per esempio **ls [q-s]*** lista i file con nomi che iniziano con un carattere compreso tra q e s

commento fino alla fine della linea

**** escape (segnala di **non interpretare** il carattere successivo come speciale)

Esempi con 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 *

Variabili nella shell

In ogni shell è possibile **definire un insieme di variabili** (trattate come stringhe) con **nome e valore**

- ❑ i riferimenti ai **valori delle variabili** si fanno con il **carattere speciale \$** (**\$nomevariabile**)

- ❑ si possono fare **assegnamenti**

nomevariabile=\$nomevariabile
l-value r-value

Esempi

- ❑ **X=2** # !!! NO spazio prima/dopo =

- ❑ **echo \$X** (visualizza 2)

- ❑ **echo \$PATH** (mostra il contenuto della variabile PATH)

- ❑ **PATH=/usr/local/bin:\$PATH** (aggiunge la directory /usr/local/bin alle directory del path di default)

Ambiente di esecuzione

- Ogni comando esegue **nell'ambiente associato (insieme di variabili di ambiente definite)** alla shell che esegue il comando
- ogni shell **eredita l'ambiente dalla shell** che l'ha creata
 - nell'ambiente ci sono **variabili** alle quali il comando può fare riferimento:
 - ❑ **variabili con significato standard: PATH, USER, TERM, ...)**
 - ❑ **variabili user-defined**

Variabili

Per vedere tutte le variabili di ambiente e i valori loro associati si può utilizzare il comando **set**:

```
BASH=/usr/bin/bash
HOME=/space/home/wwwlia/www
PATH=/usr/local/bin:/usr/bin:/bin
PPID=7497
PWD=/home/Staff/CarloGiannelli
SHELL=/usr/bin/bash
TERM=xterm
UID=1015
USER=cgiannelli
```

Espressioni

Le **variabili shell sono stringhe**. È comunque possibile **forzare l'interpretazione numerica** di stringhe che contengono la codifica di valori numerici

❑ comando **expr**:

```
expr 1 + 3
```

Esempio:

```
echo risultato: var+1
```

var+1 è il risultato della corrispondente espressione?

Espressioni

Le **variabili shell sono stringhe**. È comunque possibile **forzare l'interpretazione numerica** di stringhe che contengono la codifica di valori numerici

❑ comando **expr**:

expr 1 + 3

Esempio:

echo risultato: var+1

var+1 è il risultato della corrispondente espressione?

echo risultato: `expr \$var + 1`

a che cosa serve?

Esempio

```
#!/bin/bash
A=5
B=8
echo A=$A, B=$B
C=`expr $A + $B`
echo C=$C
```

file somma

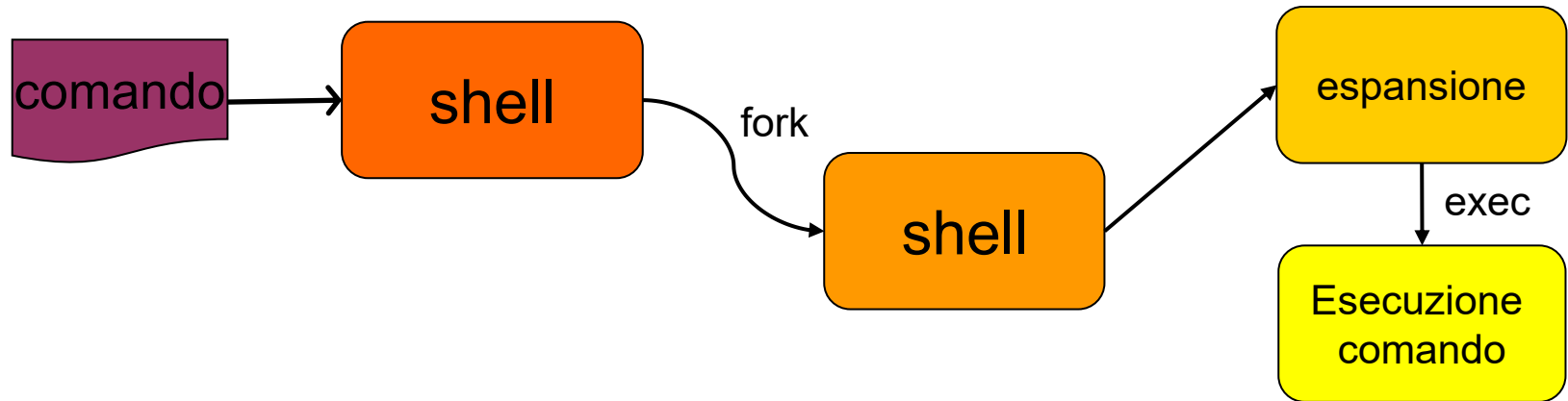
bash\$ **somma**

invocazione

A=5, B=8
C=13

output

Espansione/Parsing: sostituzione della shell



Prima della esecuzione, il comando viene scandito (**parsing**), alla ricerca di caratteri speciali (*, ?, \$, >, <, |, etc.)

- La shell **prima prepara i comandi come filtri**: ridirezione e piping di ingresso uscita
- Nelle successive scansioni, se shell trova altri caratteri speciali, **produce delle sostituzioni (passo di espansione)**

Passi di sostituzione

Sequenza dei passi di sostituzione

1) Sostituzione dei comandi

- comandi contenuti tra `` `` (**backquote**) sono eseguiti e sostituiti dal risultato prodotto

2) Sostituzione delle variabili e dei parametri

- **nomi delle variabili** (`$nome`) sono espansi nei valori corrispondenti

3) Sostituzione dei metacaratteri in nomi di file

- metacaratteri `*` `?` `[]` sono espansi nei **nomi di file** secondo un meccanismo di **pattern matching**

Inibizione dell'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
- ❑ `'` `'` (apici): proteggono da qualsiasi tipo di espansione
→ vedi 1, 2 e 3 slide precedente
- ❑ `"` `"` (doppi apici) proteggono dalle espansioni con l'eccezione di `$` `\` ``` ``` (**backquote**)
→ vedi 1 e 2 slide precedente, NO 3

Esempi inibizione dell'espansione

y=3

echo ' * e \$y ' # produce * e \$y

echo " * e \$y " # produce * e 3

echo "`pwd`" # stampa nome dir corrente

sia " che ' impediscono alla shell di interpretare i caratteri speciali per la ridirezione (< > >>) e per il piping (|)

Comando eval

```
y=3
```

```
x='$y'
```

```
echo $x    # stampa $y (perché x vale $y)
```

Per ottenere una ulteriore sostituzione, se richiesta, bisogna forzarla → comando **eval**

```
eval echo $x    # stampa 3 (perché valuta $y)
```

eval esegue il comando passato come argomento (dopo che la shell ne ha fatto il parsing).

eval consente quindi una ulteriore fase di sostituzione.

Esempio eval

Esempio:

```
$ p='ls|more'
```

```
$ $p
```

```
ls|more: command not found
```

```
$ eval $p
```

```
Ascii_pic
```

```
Bfile
```

```
...
```

Altri esempi

- `rm '$var'*`
 - ❑ Rimuove i file che cominciano con `*$var`
- `rm "$var"*`
 - ❑ Rimuove i file che cominciano con `*<contenuto della variabile var>`
- `host203-31:~ carlo$ echo "<`pwd`>"`
`</Users/carlo>`
- `host203-31:~ carlo$ echo '<`pwd`>'`
`<`pwd`>`
- `A=1+2 B=`expr 1 + 2``
 - ❑ In A viene memorizzata la stringa `1+2`, in B la stringa `3` (`expr` forza la valutazione aritmetica della stringa passata come argomento)

Riassumendo: passi successivi del parsing della shell



R ridirezione dell'input/output

`echo hello > file1` # crea `file1` e #
collega a `file1` lo stdout di `echo`

1. sostituzione dei comandi (backquote)

``pwd`` → `/temp`

2. sostituzione di variabili e parametri

`$HOME` → `/home/staff/cgiannelli`

3. sostituzione di metacaratteri

`plu?o*` → `plutone`

Altri comandi utili: cut

- **cut** è un altro comando utilissimo per la manipolazione di testo, che consente di selezionare parti di una stringa (o di ciascuna riga di un file).
- cut permette di **selezionare i caratteri della stringa** che si trovano nelle posizioni specificate (opzione -c).
- cut può **dividere una stringa in più campi** (opzione -f) usando uno specifico delimitatore (opzione -d) e può selezionare i campi desiderati.

Altri comandi utili: cut

- Esempio 1: Selezione caratteri

```
cgiannelli@linuxdid:~$ s="pippo,pluto,paperino"
```

```
# stampa dal secondo al nono carattere
```

```
cgiannelli@linuxdid:~$ echo $s | cut -c 2-9
```

```
ippo,plu
```

```
# stampa dal secondo carattere a fine riga
```

```
cgiannelli@linuxdid:~$ echo $s | cut -c 2-
```

```
ippo,pluto,paperino
```

```
# stampa il primo e il quinto carattere
```

```
cgiannelli@linuxdid:~$ echo $s | cut -c 1,5
```

```
po
```

Altri comandi utili: cut

- Esempio 2: Selezione campi

```
cgianelli@linuxdid:~$ s="pippo,pluto,paperino"
```

```
# seleziona il secondo campo della stringa dove
```

```
# i campi sono divisi da virgole
```

```
cgianelli@linuxdid:~$ echo $s|cut -f 2 -d ','  
pluto
```


Altri comandi utili: cut

- Esempio 3: Stampa il nome e la home directory di ciascun utente del sistema

```
cgiannelli@linuxdid:~$ cut -d : -f 1,6 /etc/passwd  
root:/root  
cgiannelli:/home/cgiannelli  
...
```

- Si ricordi il formato del file /etc/passwd:
username:password:UID:GID:commento:directory:shell

Altri comandi utili: tr

- **tr** è un comando che permette di fare semplici trasformazioni di caratteri all'interno di una stringa (o di ciascuna riga di un file), ad esempio sostituendo tutte le virgole con degli spazi.

- Esempi:

```
cgianelli@linuxdid:~$ str="pluto,plutone"
```

```
# sostituisce tutte le occorrenze del carattere # “,” con “:” in tutta la stringa
```

```
cgianelli@linuxdid:~$ echo $str | tr , :  
pluto:plutone
```

```
# elimina (opzione -d) il carattere “,” dalla  
# stringa
```

```
cgianelli@linuxdid:~$ echo $str | tr -d ,
```

- plutoplutone

Altri comandi utili: tr

- tr esegue anche sostituzioni tra set di caratteri.

```
# sostituisce caratteri minuscoli con maiuscoli
cgiannelli@host:~$ echo $str|tr "[:lower:]" "[:upper:]"
PLUTO,PLUTONE
```

```
cgiannelli@linuxdid:~$ str="qui,quo,qua:pluto:pippo"
# sostituisce tutte le occorrenze dei caratteri
# "," e ":" con " " e "," (rispettivamente)
# in tutta la stringa
cgiannelli@linuxdid:~$ echo $str | tr ",:;" " ,"
qui quo qua,pluto,pippo
```

Altri comandi utili: seq

- **seq** è un comando che stampa sequenze di numeri. Può essere molto utile per la realizzazione di cicli negli script.
- Esempi:

stampa numeri da 1 a 5

```
cgiannelli@linuxdid:~$ seq 1 5
```

1

2

3

4

5

Altri comandi utili: find

- **find** permette di cercare file all'interno del file system che soddisfano i requisiti specificati dall'utente, ed eventualmente di manipolarli.
- Esempio: stampa tutti i nomi delle sottodirectory contenute in \$HOME/code
cgiannelli@linuxdid:~\$ find code -type d
- Esempio: trova tutti i file *.bkp nella directory mydir ed eliminali
cgiannelli@linuxdid:~\$ find mydir -name "*.bkp" -type f -delete

Altri comandi utili: tee

- **tee** copia lo standard input sia nello standard output sia in un file passato come parametro.
- È molto utile quando si scrivono comandi con molte pipe, per monitorare il funzionamento di uno stadio della pipe.

- Esempio e opzioni:

```
cat nomefile.txt | grep stringa | tee altrofile.txt
```

```
cat nomefile.txt | grep stringa | tee -a altrofile.txt # append
```

Scripting: realizzazione file comandi

Shell: lo strumento RAD definitivo

I sistemi Unix propongono numerosissimi comandi di sistema. Attraverso l'uso in combinazione di questi comandi è possibile **realizzare in modo rapido applicazioni anche molto complesse.**

RAD: Rapid Application Development

Shell: lo strumento RAD definitivo

Niente evidenzia il valore della shell più del seguente aneddoto di rilevante importanza storica nel mondo dell'informatica.

Nel 1986, Jon Bentley, columnist dell'importantissima rivista Communications of the ACM chiese a Donald Knuth, il guru mondiale degli algoritmi, di scrivere un programma per leggere un file di testo, **determinare le N parole più ricorrenti e stampare una lista ordinata di quelle parole insieme alla rispettiva frequenza**. Knuth sviluppò il programma richiesto utilizzando il linguaggio di programmazione Pascal e lo stile literate programming. Il programma scritto da Knuth era, ovviamente, un capolavoro: **più di 10 pagine di Pascal** perfettamente commentate che facevano un uso estremamente intelligente di algoritmi e strutture dati appositamente progettate. Knuth ovviamente aveva dedicato moltissimo tempo ed energie alla scrittura del programma.

Doug McIlroy, invitato da Bentley a commentare pubblicamente il lavoro di Knuth, rispose che la stessa cosa si poteva fare con **6 righe di shell!!!**

<http://www.leancrew.com/all-this/2011/12/more-shell-less-egg/>

Una moltitudine di Shell

La shell non è unica. Nei moderni sistemi Linux (e Unix) sono disponibili diversi tipi di shell.

- sh: Bourne Shell
- bash: Bourne Again Shell (versione avanzata di sh)
- zsh: Z Shell (versione molto avanzata di sh)
- ksh: Korn Shell
- csh: C Shell (sintassi simile al C)
- tcsh: Turbo C Shell (versione avanzata di csh)
- rush: Ruby Shell (shell basata su Ruby)
- hotwire: (propone un interessante e innovativo modello integrato di terminale e shell)

Una moltitudine di Shell

Ogni utente può specificare quale shell desidera usare.

La shell più usata è sicuramente Bash, che è molto simile alla shell di Bourne (/bin/sh).

- La shell di Bourne, creata da Stephen Bourne dei laboratori AT&T negli anni '70, ha sostituito la shell di Thompson, che non consentiva lo sviluppo di file comandi.
- La shell di Bourne è stata poi affiancata da altre shell come Bash, Korn shell e Z shell, che ne riprendono la sintassi, ampliandola significativamente.

Si ricordi che, come i nomi di file nel filesystem, in ambiente Unix anche la shell è case sensitive.

File comandi

Shell è un **processore comandi** in grado di interpretare **file sorgenti in formato testo e contenenti comandi** → **file comandi (script)**

Linguaggio comandi (vero e proprio linguaggio programmazione)

- ❑ Un file comandi può comprendere
 - **statement per il controllo di flusso**
 - **variabili**
 - **passaggio dei parametri**

NB:

- ❑ **Quali statement** sono disponibili dipende da **quale shell** si utilizza
- ❑ File comandi viene **interpretato** (non esiste una fase di compilazione)
- ❑ File **comandi deve essere eseguibile** (usare **chmod**)

Scelta della shell

La prima riga di un file comandi deve specificare **quale shell si vuole utilizzare**: **`#! <shell voluta>`**

- ❑ Es: **`#! /bin/bash`**
- ❑ **`#`** è visto dalla shell come un commento ma...
- ❑ **`#!`** è visto da SO come identificatore di un file di script
SO capisce così che l'interprete per questo script sarà **`/bin/bash`**

- Se questa riga è assente viene scelta la shell di preferenza dell'utente

File comandi

È possibile memorizzare **sequenze di comandi**
all'interno di file eseguibili:

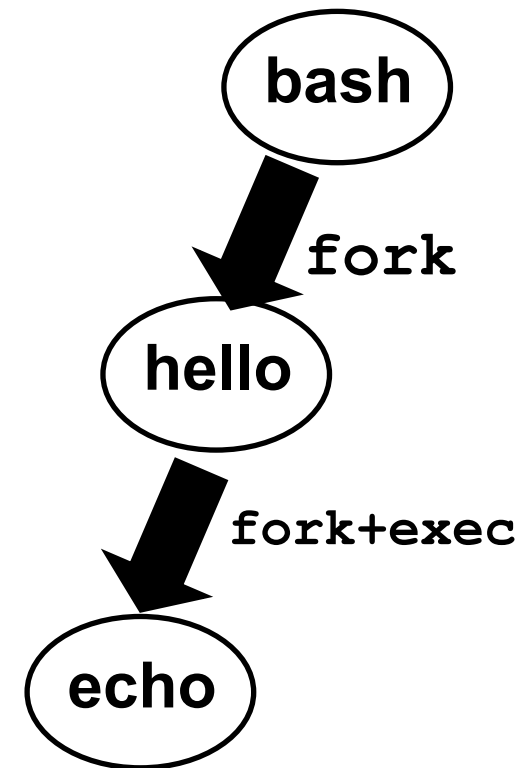
file comandi (script)

Ad esempio:

```
#!/bin/bash  
echo hello world!
```

file hello

```
bash$ hello  
hello world!
```



Passaggio parametri

`./nomefilecomandi arg1 arg2 ... argN`

Gli argomenti sono ***variabili posizionali*** nella linea di invocazione contenute nell'ambiente della shell

- **\$0** rappresenta il comando stesso
- **\$1** rappresenta il primo argomento ...
- è possibile far scorrere tutti gli argomenti verso sinistra
→ **shift**

\$0 non va perso, solo gli altri sono spostati (\$1 perso)

	\$0	\$1	\$2
prima di shift	DIR	-w	/usr/bin
dopo shift	DIR	/usr/bin	

- È possibile riassegnare gli argomenti → **set**
 - ❑ **set exp1 exp2 exp3 ...**
 - ❑ gli argomenti sono assegnati secondo la posizione

Altre informazioni utili

Oltre agli argomenti di invocazione del comando

- ❑ **\$*** insieme di **tutte le variabili posizionali**, che corrispondono arg del comando: \$1, \$2, ecc.
- ❑ **\$#** **numero di argomenti** passati (**\$0 escluso**)
- ❑ **\$?** valore (int) restituito dall'ultimo comando eseguito
- ❑ **\$\$** id numerico del processo in esecuzione (pid)

Semplici forme di input/output

- ❑ `read var1 var2 var3` #input
- ❑ `echo var1 vale $var1 e var2 $var2` #output
 - **read** la stringa in ingresso viene attribuita alla/e variabile/i secondo corrispondenza posizionale

Strutture di controllo

Ogni comando in uscita restituisce un **valore di stato**, che indica il suo **completamento o fallimento**

Tale valore di uscita è posto nella variabile ?

- ❑ `$?` può essere riutilizzato in espressioni o per controllo di flusso successivo

Stato vale usualmente:

- ❑ zero: comando OK
- ❑ valore positivo: errore

Esempio

```
host203-31:~ carlo$ cp a.com b.com
```

```
cp: cannot access a.com
```

```
host203-31:~ carlo$ echo $?
```

```
2
```

test

Comando per la **valutazione di una espressione**

- ❑ **test -<opzioni> <nomefile>**

Restituisce uno stato uguale o diverso da zero

- ❑ valore **zero** → **true**
- ❑ valore **non-zero** → **false**

ATTENZIONE: convenzione opposta rispetto al linguaggio C!

- ❑ Motivo: i codici di errore possono essere più di uno e avere significati diversi

Alcuni tipi di test

test

- ❑ `-f <nomefile>` esistenza di file
- ❑ `-d <nomefile>` esistenza di direttori
- ❑ `-r <nomefile>` diritto di lettura sul file (**-w** e **-x**)
- ❑ `test <stringa1> = <stringa2>` uguaglianza stringhe
- ❑ `test <stringa1> != <stringa2>` diversità stringhe

ATTENZIONE:

- Gli **spazi intorno a** `=` (o `a !=`) sono **necessari**
- `stringa1` e `stringa2` possono contenere metacaratteri (attenzione alle espansioni)
- ❑ `test -z <stringa>` vero se **stringa nulla**
- ❑ `test <stringa>` vero se **stringa non nulla**

Strutture di controllo: alternativa

```
if <lista-comandi>
then
    <comandi>
[elif <lista_comandi>
then <comandi>]
[else <comandi>]
fi
```

ATTENZIONE:

- ❑ Le parole chiave (do, then, fi, ...) devono essere o a capo o dopo il separatore ;
- ❑ if controlla il valore in uscita **dall'ultimo comando di <lista-comandi>**

Esempio

```
# fileinutile
# risponde "sì" se invocato con "sì" e un numero
  <= 24
if test $1 = sì -a $2 -le 24
  then echo sì
  else echo no
fi
```

```
#test su argomenti
if test $1; then echo OK
  else echo Almeno un argomento
fi
```

Alternativa multipla

```
# alternativa multipla sul valore di var
case <var> in
    <pattern-1>)
        <comandi>;
    ...
    <pattern-i> | <pattern-j> | <pattern-k>)
        <comandi>;
    ...
    <pattern-n>)
        <comandi> ;;
esac
```

Importante: nell'alternativa multipla si possono usare metacaratteri per fare pattern-matching (non sono i “soliti” metacaratteri su nome di file)

Esempi

```
read  risposta
case  $risposta in
    S* | s* | Y* | y* ) <OK>;
    * ) <problema>;
esac
```

```
# append: invocazione  append [dadove] adove
case $# in
    1) cat >> $1;;
    2) cat < $1 >> $2;;
    *) echo  uso: append [dadove] adove;
       exit 1;;
esac
```

Cicli enumerativi

```
for <var> [in <list>] # list=lista di stringhe
do
    <comandi>
done
```

- Scansione della lista <list> e **ripetizione del ciclo per ogni stringa presente nella lista**
- Scrivendo solo **for i** si itera con valori di **i in \$***

Esempi

- `for i in *`
 - ❑ esegue per tutti i file nel direttorio corrente
- `for i in `ls s*``
`do <comandi>`
`done`
- `for i in `cat file1``
`do <comandi per ogni parola del file file1>`
`done`
- `for i in 0 1 2 3 4 5 6`
`do`
`echo $i`
`done`

Ripetizioni non enumerative

```
while <lista-comandi>  
do  
    <comandi>  
done
```

Si ripete per tutto il tempo che il valore di stato dell'ultimo comando della lista è zero (successo)

```
until <lista-comandi>  
do  
    <comandi>  
done
```

Come while, ma inverte la condizione

Uscite anomale

- ❑ vedi C: **continue**, **break** e **return**
- ❑ **exit [status]**: system call di UNIX, anche comando di shell

Menu testuali

Il comando **select** può essere usato per realizzare semplici menù testuali.

Esempio:

```
select risposta in pippo pluto paperino;  
do  
    echo "Hai scelto $risposta"  
done
```

All'esecuzione dello script di cui sopra, la shell stampa il menù:

```
1) pippo  
2) pluto  
3) paperino  
#?
```

Nel caso l'utente digiti 1 la shell stampa:

Hai scelto pippo.

Esempi uso select

Il comando **select** è particolarmente utile per la selezione di file:

```
#!/bin/bash
echo "Scegli quale file cancellare"
select i in `ls *.bak`
do
    echo "Selezionato: $i"
    rm -f $i
done
```

Esempi uso select

Altra utile applicazione costituita dalla selezione tra un insieme di operazioni:

```
#!/bin/bash
select i in "stampa" "cancella"
do
    echo "Selezionato: $i"
done
if test $i = "stampa"
then
    ...
fi
...
```

Ambiente di un file comandi

L'ambiente dei file comandi è composto da:

- variabili predefinite
- direttorio corrente
- insieme di variabili usate e variate dall'utente

L'ambiente è accessibile agli shell figli, che possono accrescere il tutto e aggiungere nuove variabili

- viene passato solo l'ambiente predefinito iniziale
- sono aggiunti i valori delle sole variabili esplicitamente esportate (istruzione export)

NB: in caso di modifica di variabili predefinite, i nuovi valori devono essere esplicitamente esportati

Ambiente di un file comandi

L'ambiente padre è preservato

OGNI COMANDO è ESEGUITO da una NUOVA SHELL,
distinta dalla shell-padre.

La shell-figlio ottiene una copia (parziale) dell'ambiente del
padre ma **le eventuali modifiche NON sono viste dalla
shell padre.**

Ambiente di un file comandi: esempio 1

```
# file view filename [filename]
case $# in
    0) echo Usage is: $0 filename [filename]
        exit 1;;
    *) ;;
esac
for i in $*                # in $* poteva essere omissso
do
    if test -f $i          # se il file esiste
    then
        echo $i           # visualizza il nome del file
        cat $i            # visualizza il contenuto del file
    else echo file $i non presente
    fi
done
```


Ambiente di un file comandi: esempio 2

```
# file lista <filenames>
for i
do
    echo -n "$i ?" > /dev/tty          # terminale actual. connesso
    # -n = non emettere newline;
    # ? non è sostituito (ci sono le virgolette)
    read answer
    case $answer in
        y*| Y*| s* | S* ) echo $i; cat $i ;;
    esac
done
```

NOTA BENE: I comandi utente sono trattati in modo OMOGENEO rispetto ai comandi di sistema

```
lista f* > temp
```

```
lista ?p* | grep <stringa>
```

Ambiente di un file comandi: esempio 3

```
# file cercadir [direttorioassoluto] file
```

```
case $# in
```

```
    0) echo errore. Usa $0 [direttorio] file
```

```
    exit 2 ;;
```

```
    1) d=`pwd`; f=$1;;
```

```
    2) d=$1;    f=$2;;
```

```
esac
```

```
PATH= ...      # quali direttori bisogna considerare?
```

```
                # PATH deve comprendere il direttorio in cui
```

```
                # si trova cercadir stesso
```

```
export PATH
```

Ambiente di un file comandi: esempio 3

```
cd $d
if test -f $f
    then echo il file $f è in $d
fi

for i in *
do
    if test -d $i
        then echo direttorio $d/$i
            cercadir `pwd`/$i $f      # ricorsione
        fi
done
```

Ambiente di un file comandi: esempio 3

NB: questo esempio sottolinea il tipico problema che c'è spostandosi nei direttori: i comandi non inclusi nel path non verranno trovati

In alternativa si possono considerare i **nomi assoluti** dei comandi che non sono nel path

File comandi ricorsivi tendono ad agire su un sottodirettorio per volta e a lanciare una invocazione per ogni sottodirettorio trovato: non si devono prevedere così le profondità dell'albero dei direttori

Controllo degli argomenti

Il controllo degli argomenti nei FILE COMANDI è fondamentale

È necessario verificare gli argomenti

- devono essere innanzitutto nel numero giusto
- poi del tipo richiesto

invocazione di comando per ...

case \$# in

0|1|2|3) echo Errore. Almeno 4 argomenti ... >& 2

exit 1;;

esac

in alternativa:

if test \$# -lt 4

echo Errore ...

fi

Controllo degli argomenti

in caso di argomenti corretti:

primo argomento: dev'essere un direttorio (o file)

if test ! -d \$1

then echo argomento sbagliato: \$1 direttorio >& 2; exit 2

fi

primo argomento: se è un nome di file assoluto

case \$1 in

/*) if test ! -f \$1

then echo argomento sbagliato: \$1 file ; exit 2

fi;;

*) echo argomento sbagliato: \$1 assoluto; exit 3;;

esac

Controllo degli argomenti

primo argomento: se è un nome di file relativo

case \$1 in

*/) echo argomento sbagliato: \$1 nome relativo ; exit 3;;

*) ;;

esac

secondo argomento: dev'essere stringa numerica

si potrebbe usare un case con match [0-9] ?

e case con match [0-9] | [0-9][0-9] ?

Si veda il comando cut che consente di selezionare le colonne delle

righe di un file. Si metta in pipe l'argomento e lo si selezioni in un ciclo

una lettera alla volta: echo \$2 | cut -d\$i

Il singolo carattere può essere estratto ed esaminato

Controllo degli argomenti

```
# il comando expr può verificare una espressione numerica
# (è necessaria l'operazione)
expr $i + 0 > /dev/null 2> /dev/null
if test $? -ne 0
    then echo errore in argomento numerico: $i ; exit 4
fi
# qual è la ragione delle ridirezioni su /dev/null?
```


Controllo degli argomenti

Si possono eliminare alcuni argomenti per comodità di scansione: si usi lo shift, dopo avere salvato gli argomenti che vengono eliminati

```
salva1=$1 # salvataggio di $1
```

```
shift
```

```
salva2=$1 # salvataggio di $2
```

```
shift
```

```
# cosa vale adesso $*?
```

```
for i
```

```
do
```

```
    # il ciclo è fatto per tutti gli argomenti esclusi quelli tolti
```

```
done
```

```
# gli argomenti iniziali sono dati da $salva1 $salva2 $*
```