

# Unix e Bash

*Introduzione, di cosa ci occuperemo*

# Obiettivi

- Illustrare le funzionalità principali della shell di Unix e di tool comunemente presenti nelle distribuzioni Unix (ref. Linux + Bash)
- Descrivere l'iso *interattivo* della shell
  - Concetti base, interazione, comandi, personalizzazione
- Struttura interna della shell:
  - variabili, espansione della riga di comando, ridirezione, comandi composti (liste, pipe, sequenze condizionali)
- Programmare la shell (*shell scripting*)
  - Funzioni, costrutti di controllo, debugging

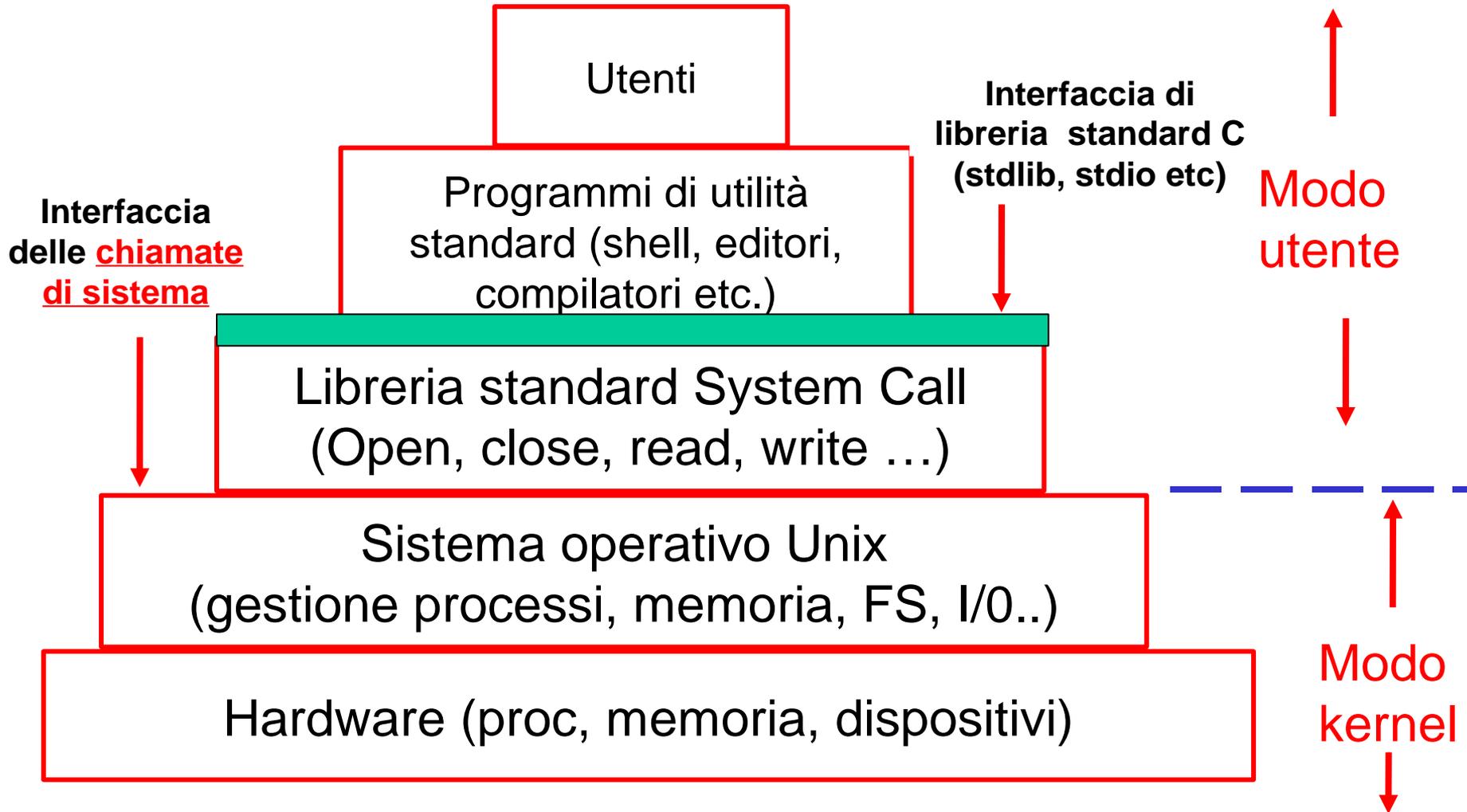
# La shell di Unix

Cos'è, vari tipi di shell, la Bash  
(Bourne Again SHell), il punto di vista  
dell'utente ...

# Cos'è Unix/Linux?

- Unix è un sistema operativo
  - Gestore/virtualizzatore di risorse
- é anche un ambiente di sviluppo sw
  - Fornisce utilità di sistema, editor, compilatori, assembleri, debugger etc.
- Linux è un SO `POSIX compatibile'
  - Fornisce le funzionalità tipiche dei sistemi Unix
  - è *open source* ovvero:
    - Il suo codice è disponibile, può essere modificato e ridistribuito

# UNIX/Linux: struttura generale



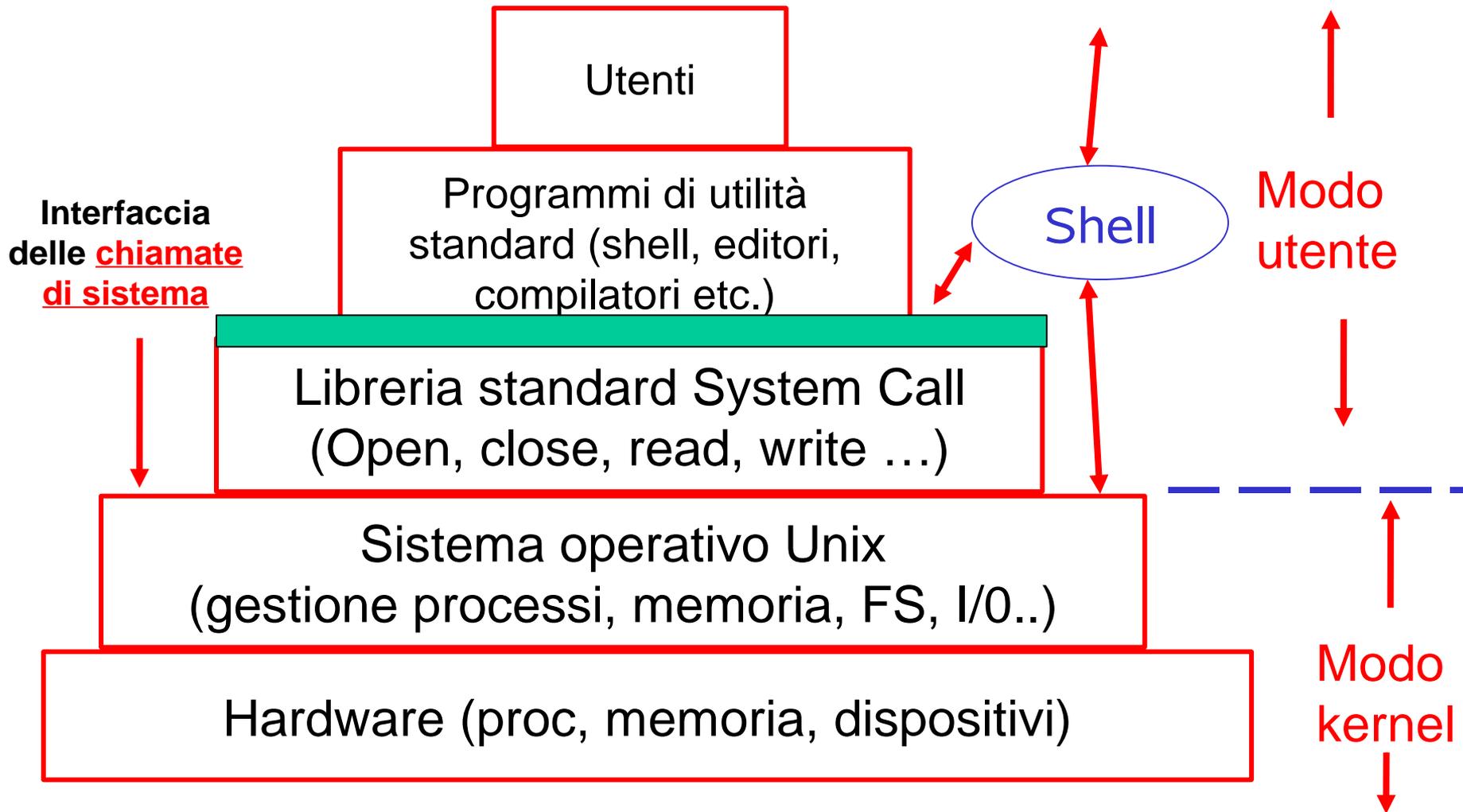
# La filosofia dell'ambiente Unix

- Si rivolge a programmatori
- Comandi con sintassi minimale testuale
  - Diminuisce i tempi di battitura
- Ogni componente/programma/tool realizza una sola funzione in modo semplice ed efficiente
  - es. **who** e **sort**
- Più componenti si possono legare per creare un'applicazione più complessa:
  - es. **who | sort**

# La filosofia dell'ambiente Unix (2)

- Il principale software di interfaccia è la shell
  - un normale programma senza privilegi speciali
  - gira in spazio utente
  - interfaccia testuale
- è usata spesso come supporto per automatizzare attività di routine
  - programmazione di shell (shell scripting)

# UNIX/Linux: shell (2)



# Cos'è una shell .....

- è un normale programma!
- è un *interprete di comandi*
  - funziona in modo interattivo e non interattivo
  - Nella versione interattiva: fornisce una interfaccia testuale per richiedere comandi

**bash: ~\$**                    **-- (prompt) nuovo comando?**

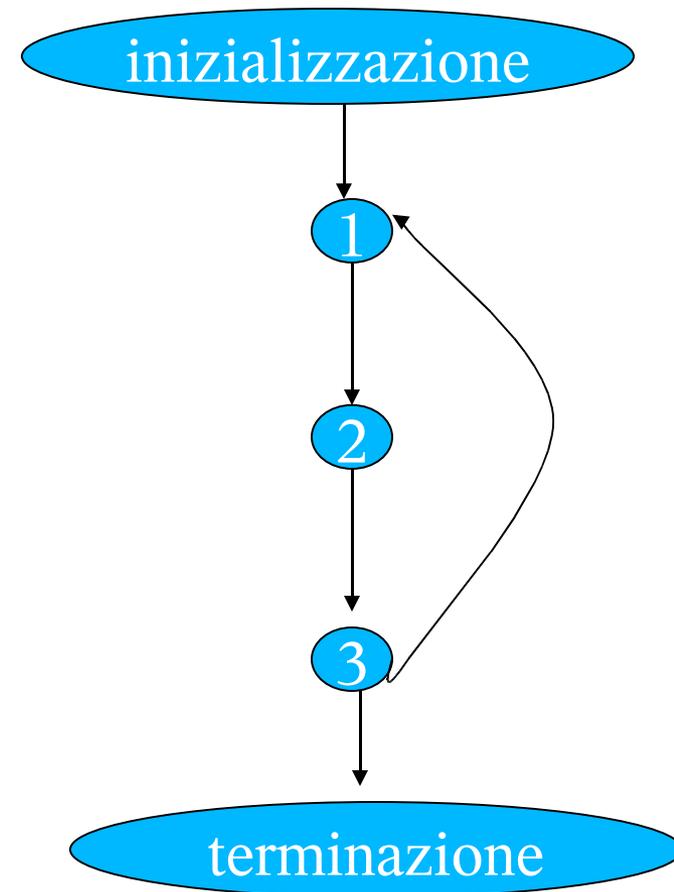
**bash: ~\$ date**                **-- l'utente da il comando**

**Thu Mar 12 10:34:50 CET 2005**    **-- esecuzione**

**bash: ~\$**                    **-- (prompt) nuovo comando?**

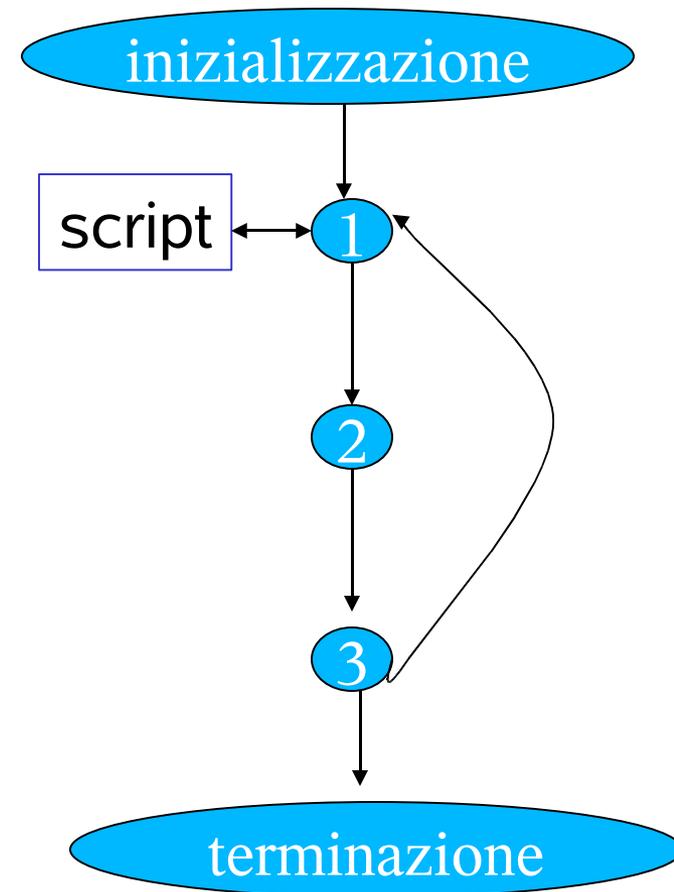
# Cos'è una shell ..... (2)

- Ciclo di funzionamento shell interattiva:
  - *inizializzazione*
  - *ciclo principale*
    1. Richiede un nuovo comando (prompt)
    2. L'utente digita il comando
    3. La shell interpreta la richiesta e la esegue
  - *termina con exit oppure EOF*



# Cos'è una shell ..... (3)

- Funzionamento non interattivo
  - comandi in un file (lo *script* )
- Ciclo:
  - *inizializzazione*
  - *ciclo principale*
    1. Legge un nuovo comando da file
    2. Lo decodifica
    3. Lo esegue
  - *termina con exit oppure EOF*



# Perché mai usare una *shell* testuale?

- Ormai tutti i sistemi Unix hanno un'interfaccia grafica, degli ambienti integrati ....
  - perché usare i comandi in linea?
- Alcuni motivi:
  - per capire come funziona 'sotto'
  - perché i comandi Unix sono stati progettati per essere semplici da scrivere ed usare su questo tipo di interfacce
  - perché una volta imparati sono più veloci e flessibili dei menu fissi e delle finestre di dialogo
  - perché sono veloci ed accessibili da remoto

# Perché mai usare una *shell* testuale? (2)

*A programmer needs a servant, not a nanny.*

A. Tanenbaum

# Cos'è una shell ..... (4)

- Ci sono vari tipi di shell
  - C shell (**cs**h, **tc**sh), Bourne shell (**sh**), Bourne Again shell (**ba**sh)
- c'è un insieme di comportamenti, funzionalità comuni
  - ognuna ha il suo linguaggio di programmazione
    - linguaggio di scripting
  - *script*: programma interpretabile da shell
    - serie di comandi salvata su file
    - combinati usando costrutti di controllo anche complessi di tipo IF, WHILE etc.
    - più altro....

# Perché programmare la shell?

- Perché non usare C/Java?
  - I linguaggi di shell sono fatti apposta per manipolare file e processi etc in ambiente Unix, cosa che li rende particolarmente adatti per automatizzare task (specie ripetitivi) in questo ambiente.
    - Es. tutti le varie distribuzioni Linux usano script bash
- Noi descriveremo le caratteristiche principali della shell Bash

# Nota: la *Bash* al *cli*

- *Bash* è oramai lo standard di fatto delle distribuzioni Linux Unix sia free che non
- per ragioni ‘storiche’ il nostro cdc continua ad usare la vecchia C shell
  - versione **tcsh** migliorata ma tant’è ...
- come sapere la vostra shell di login

```
bash:~$ echo $SHELL
```

```
tcsh
```

```
bash:~$
```

# Nota: la *bash* al cli (2)

- Come cambiare temporaneamente shell

```
(susanna) tcsh => bash
```

```
bash:~$      -- da qua lavoro con bash
```

```
...
```

```
bash:~$ exit
```

```
(susanna) tcsh =>
```

- Come cambiare shell permanentemente?

- Non è permesso in modo diretto
- NOTA: è fortemente sconsigliato dai sistemisti girarci attorno inserendo un **exec bash** negli script di inizializzazione **.cshrc** o **.login**

# Nota: la *bash* al *cli* (3)

- File di inizializzazione

- .bashrc**

- .bash\_profile**

- .bash\_aliases**

- servono per avere un ambiente di lavoro più gradevole e per personalizzare la shell interattiva (lo vedremo più avanti)
    - non ci sono sul vostro account!
    - scaricateli dal sito del corso seguendo le istruzioni

# Comandi base di Unix

- Sintassi tipica:

**nome <opzioni> <argomenti>**

- es:

***-- trovare la data***

**bash:~\$ date *--no arguments***

**Thu Mar 12 10:34:50 CET 2005**

**bash:~\$**

***-- l'utility man***

**bash:~\$ man sort** ***--one argument***

SORT(1) User Commands SORT(1)

## **NAME**

sort - sort lines of text files

## **SYNOPSIS**

sort [ OPTION ] ... [ FILE ] ..

## **DESCRIPTION**

...

**-f --ignore-case**

fold lower case to upper case characters

**RETURNS ...**

**REPORTING BUGS...**

**SEE ALSO**

Textinfo info sort ***--da emacs***

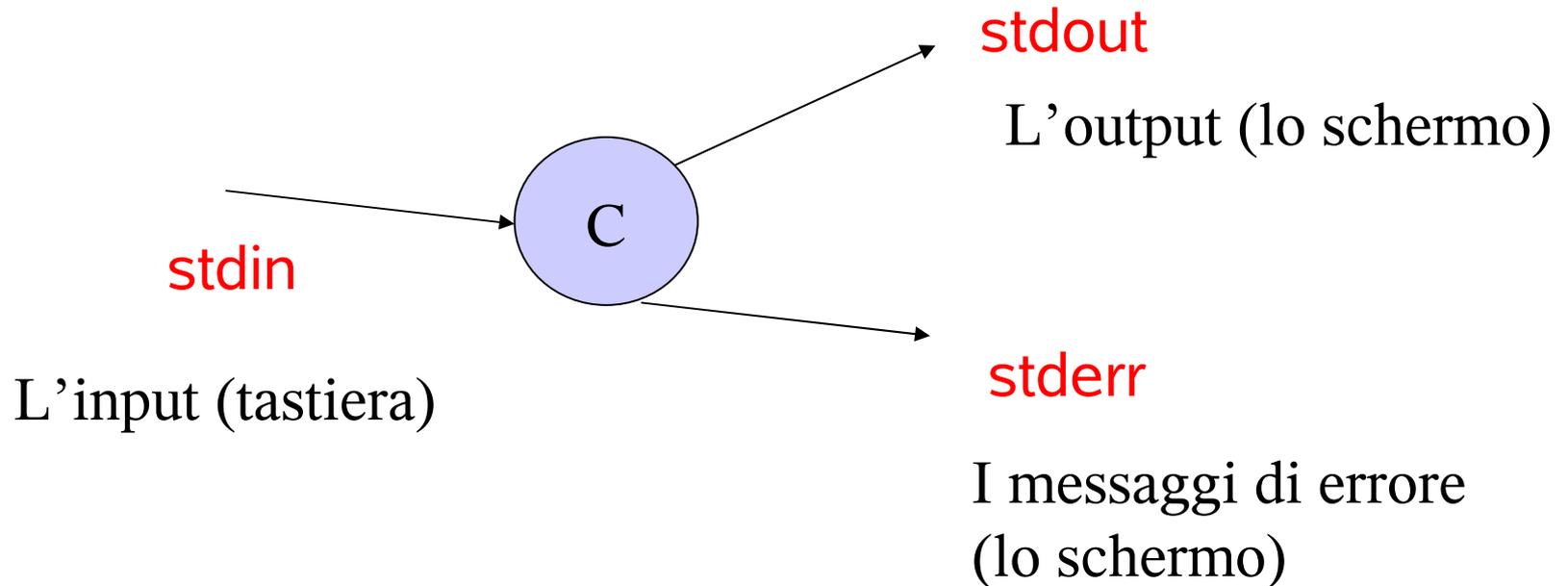
**bash:~\$**

# Standard input, output ed error

- Negli esempi visti l'output viene scritto sempre su terminale
- Ogni comando o programma che gira sotto Unix ha sempre tre canali di I/O attivi:
  - **stdin** lo standard input
  - **stdout** lo standard output
  - **stderr** lo standard error
- Di default questi tre canali sono il terminale di controllo del processo che sta eseguendo programma o il comando
  - meccanismo semplice e flessibile (ridirezione, vedi poi)

# Standard input, output ed error (2)

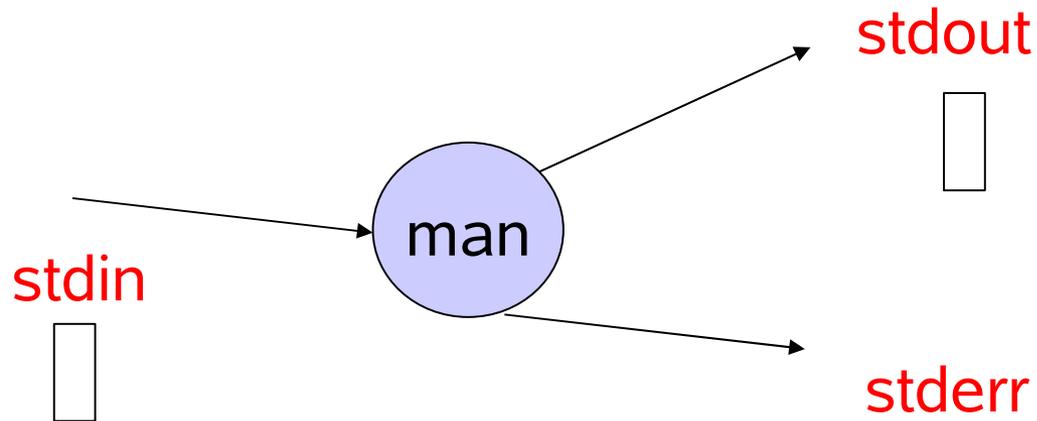
– Tipicamente



# Standard input, output and error (3)

– Es.

```
bash:~$ man ciccio
```



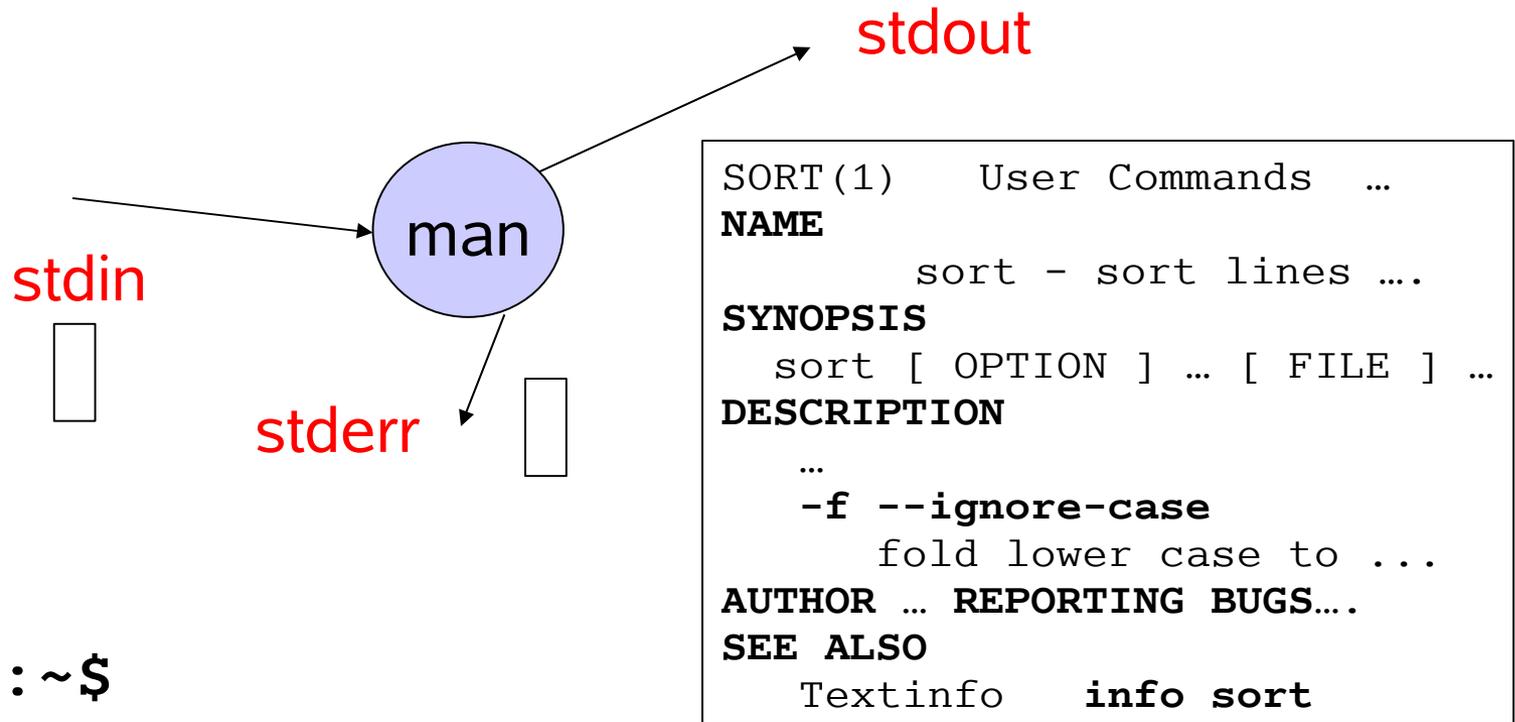
```
No manual entry for ciccio
```

```
bash:~$
```

# Standard input, output and error (4)

– Es.

```
bash:~$ man sort
```



```
bash:~$
```

# Standard input, output ed error (5)

```
bash:~$ sort
```

```
    -- attende input dall'utente
```

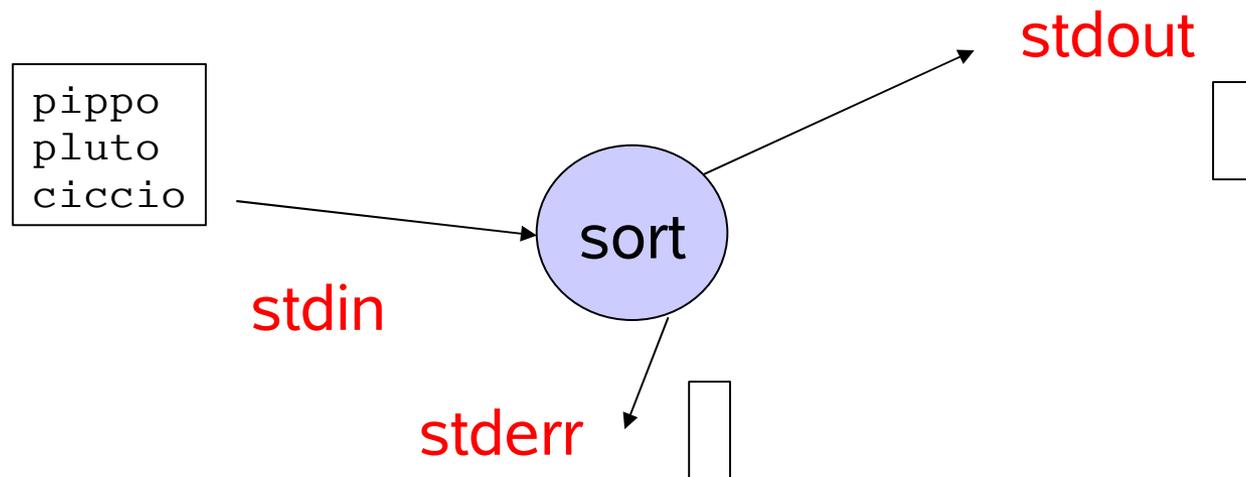
# Standard input, output and error (6)

```
bash:~$ sort
```

```
pippo
```

```
pluto
```

```
ciccio
```



# Standard input, output and error (7)

```
bash:~$ sort
```

```
pippo
```

```
pluto
```

```
ciccio
```

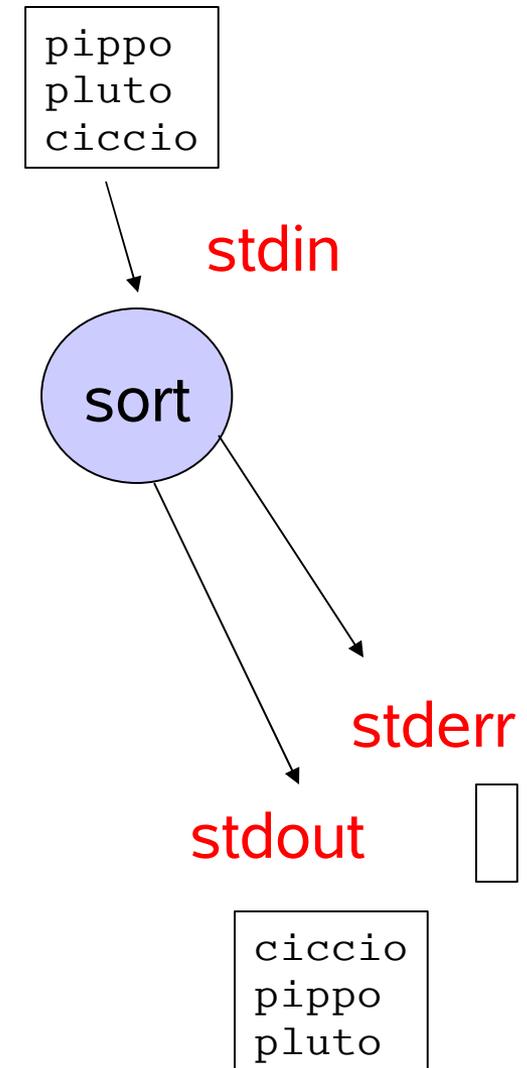
```
CTRL-d      -- EOF
```

```
ciccio
```

```
pippo
```

```
pluto
```

```
bash:~$
```



*-- Ultimi esempi: utilities with both option(s)  
and argument(s)*

*-- ricerca le descrizioni di pagine di manuale  
che contengono "printf"*

```
bash:~$ man -k printf
```

```
vasprintf (3)      - print to allocated string
```

```
vasnprintf (3)    - ...
```

```
.....
```

```
sprintf (3)       - formatted output conversion
```

```
bash:~$
```

*-- Ultimi esempi: utilities with both option(s) and argument (2)*

*-- compilazione programmi C da linea di comando*

*-- NB: comando su più linee con backslash (\)*

```
bash:~$ gcc -Wall -pedantic -o name_exe \  
source.c
```

```
bash:~$
```

*-- oggetto e debug info*

```
bash:~$ gcc -Wall -pedantic -c -g source.c
```

```
bash:~$
```

# Shell: metacaratteri

- Sono caratteri che la shell interpreta in modo ‘speciale’
  - CTRL-d, CTRL-c, &, >, >>, <, ~, |, \*....
- Forniscono alla shell indicazioni su come comportarsi nella fase di interpretazione del comando
  - li descriveremo man mano

# Alcuni comandi base

<code>ls</code>	<i>-- file listing</i>
<code>more, less, cat</code>	<i>-- contenuto del file</i>
<code>cp, mv</code>	<i>-- copia sposta file e dir</i>
<code>mkdir</code>	<i>-- crea una nuova directory</i>
<code>rm, rmdir</code>	<i>-- rimuove file, directory</i>
<code>head, tail</code>	<i>-- selez. linee all'inizio (fine) di un file</i>
<code>file, type...</code>	<i>-- tipo di un file</i>
<code>wc</code>	<i>-- conta parole, linee caratteri</i>
<code>lpr</code>	<i>-- stampa</i>

# Esempi: wc

```
bash:~$ wc -c file          -- conta caratteri
 2281 file
bash:~$ wc -l file          -- conta linee
  73 file
bash:~$ wc -w file          -- conta parole
 312 file
bash:~$ wc file             -- conta tutto
  73   312  2281 file
bash:~$
```

# Esempi: cat, file

*-- concatena il contenuto di file1 e file 2 e lo mostra su stdout*

```
bash:~$ cat file1 file2
```

*-- tipo di file*

```
bash:~$ file a.out
```

```
a.out: ELF 32-bit LSB executable Intel 80386  
version 1 (SYSV), GNU/Linux 2.2.0, dynamically  
linked (uses shared libs), not stripped
```

```
bash:~$ file pippo.n
```

```
pippo.n: ASCII text
```

```
bash:~$
```

# Editing della linea di comando

- Molte shell, fra cui la bash, offrono funzioni di *editing della linea di comando* “ereditate” da emacs. Ecco le più utili:

**CTRL-a**            --va a inizio riga

**CTRL-e**            --va a fine riga

**CTRL-k**            --cancella fino a fine linea

**CTRL-y**            --reinserisce la stringa cancellata

**CTRL-d**            --cancella il carattere sul cursore

# History

- La shell inoltre registra i comandi inseriti dall'utente. È possibile visualizzarli....

```
bash:~$ history
```

```
68 gcc main.c
```

```
69 a.out data
```

```
70 ls
```

```
71 history
```

```
bash:~$
```

# History (2)

- ... oppure richiamarli

```
bash:~$ history
```

```
68 gcc main.c
```

```
69 a.out data
```

```
70 ls
```

```
71 history
```

```
bash:~$ !l      -- l'ultimo che inizia per 'l'
```

```
ls
```

```
main.c a.out data
```

```
bash:~$ !68    -- il numero 68
```

```
gcc main.c
```

```
bash:~$
```

# History (3)

- ... un altro esempio

```
bash:~$ ls
```

```
main.c a.out data
```

```
bash:~$!!
```

*-- l'ultimo comando eseguito*

```
ls
```

```
main.c a.out data
```

```
bash:~$
```

- è possibile anche navigare su e giù per la history con le freccette (↑↓)

# Completamento dei comandi

- Un'altra caratteristica tipica delle shell è la possibilità di completare automaticamente le linee di comando usando il tasto TAB

```
bash:~$ ls
```

```
un_file_con_un_nome_molto_lungo
```

*-- voglio copiarlo sul file 'a'*

```
bash:~$ cp un TAB
```

*-- la shell completa*

```
bash:~$ cp un_file_con_un_nome_molto_lungo
```

*-- poi posso digitare 'a'*

```
bash:~$ cp un_file_con_un_nome_molto_lungo a
```

```
bash:~$
```

# Completamento dei comandi (2)

- se esistono più completamenti possibili
  - premendo `TAB` viene emesso un segnale sonoro.
  - Premendolo nuovamente si ottiene la lista di tutti i file che iniziano con il prefisso già digitato.

```
bash:~$ ls
```

```
un_file  uno.c  ns2.h
```

```
bash:~$ cp un TAB TAB
```

```
un_file  uno.c
```

```
bash:~$ cp un
```

File, directory, e directory  
corrente

# I file di Unix

- Tipi di file Unix :
  - *regular* (-): collezione di byte non strutturata
  - *directory* (**d**) : directory
  - *buffered special file* (**b**) : file che rappresenta una periferica con interfaccia a blocchi
  - *unbuffered special file* (**c**) : file che rappresenta una periferica con interfaccia a caratteri
  - *link simbolico* (**l**) : file che rappresenta un nome alternativo per un altro file X, ogni accesso a questo file viene ridiretto in un accesso a X

# I file di Unix (2)

- Tipi di file Unix (cont.):
  - *pipe* (**p**): file che rappresenta una pipe
  - *socket* (**s**) : file che rappresenta un socket

# Attributi di un file Unix

- File = dati + attributi
- Alcuni attributi dei file unix :

```
bash:~$ ls -l pippo.c
```

```
-rw-r--r-- 1 susanna users 1064 Feb 6 2005 pippo.c
```

Tipo del file  
(regolare, -)

Nome del file  
(unico all'interno  
della directory,  
case sensitive)

# Attributi di un file Unix (2)

- File = dati + attributi
- Alcuni attributi dei file unix :

```
bash:~$ ls -l pippo.c
```

```
-rw-r--r-- 1 susanna users 1064 Feb 6 2005 pippo.c
```

## Protezione

r - permesso di lettura (directory, listing)

w- permesso di scrittura (directory, aggiungere file)

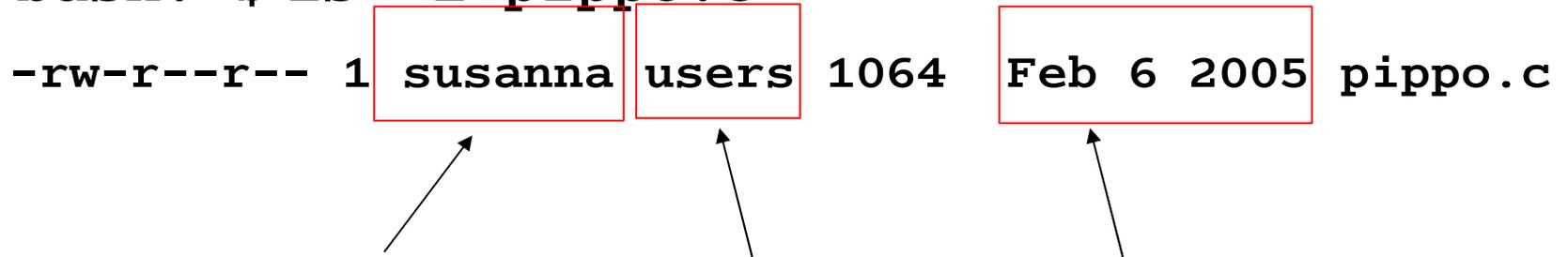
x - permesso di esecuzione (directory, accesso)

# Attributi di un file Unix (3)

- File = dati + attributi
- Alcuni attributi dei file unix :

```
bash:~$ ls -l pippo.c
```

```
-rw-r--r-- 1 susanna users 1064 Feb 6 2005 pippo.c
```



Proprietario del file

Gruppo

Data ultima modifica

# Attributi di un file Unix (4)

- File = dati + attributi
- Alcuni attributi dei file unix :

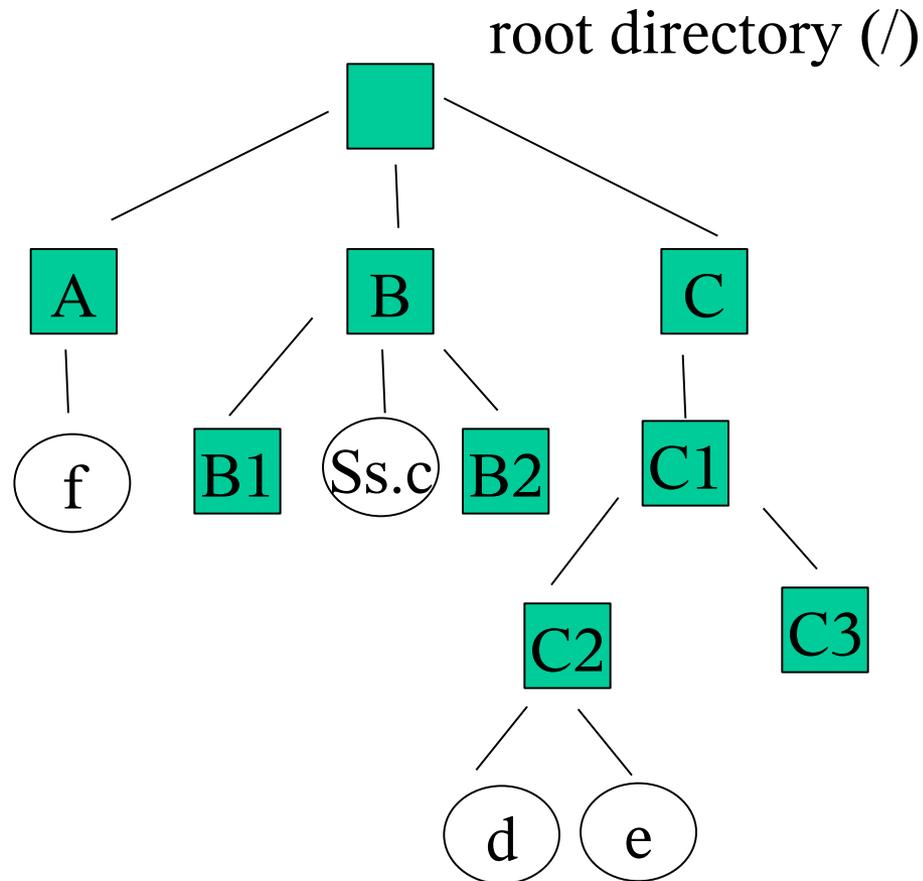
```
bash:~$ ls -l pippo.c
```

```
-rw-r--r-- 1 susanna users 1064 Feb 6 2005 pippo.c
```

Numero di blocchi su disco utilizzati

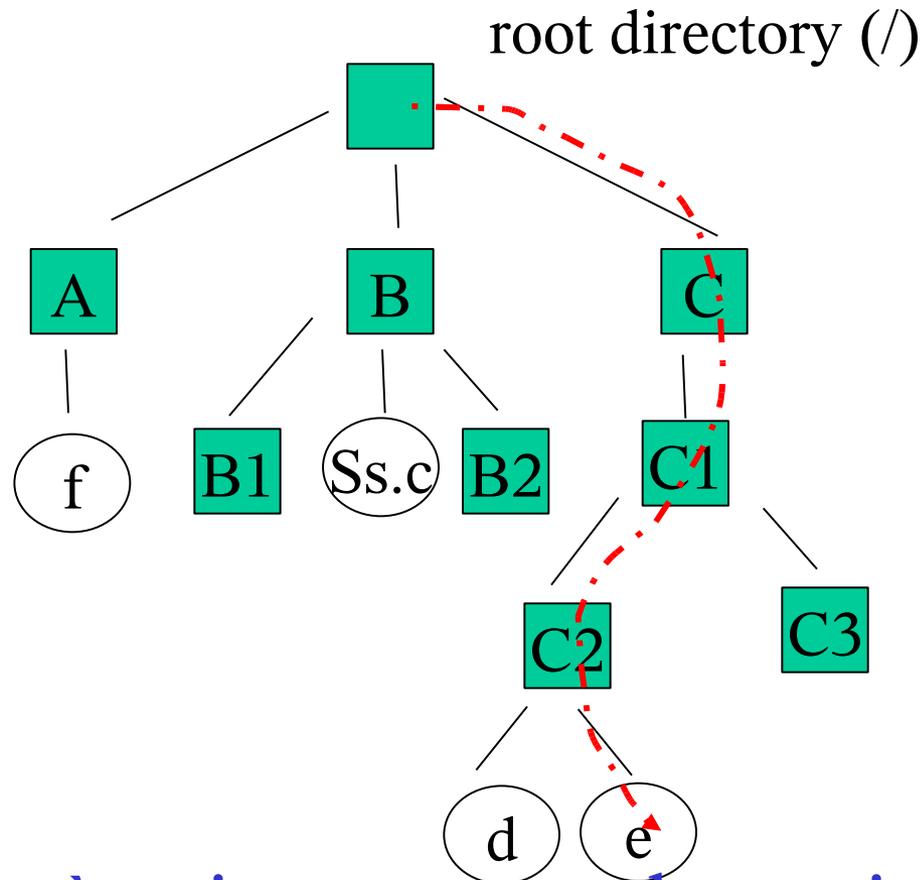
Lunghezza in byte  
del file

# Il FS di Unix è gerarchico



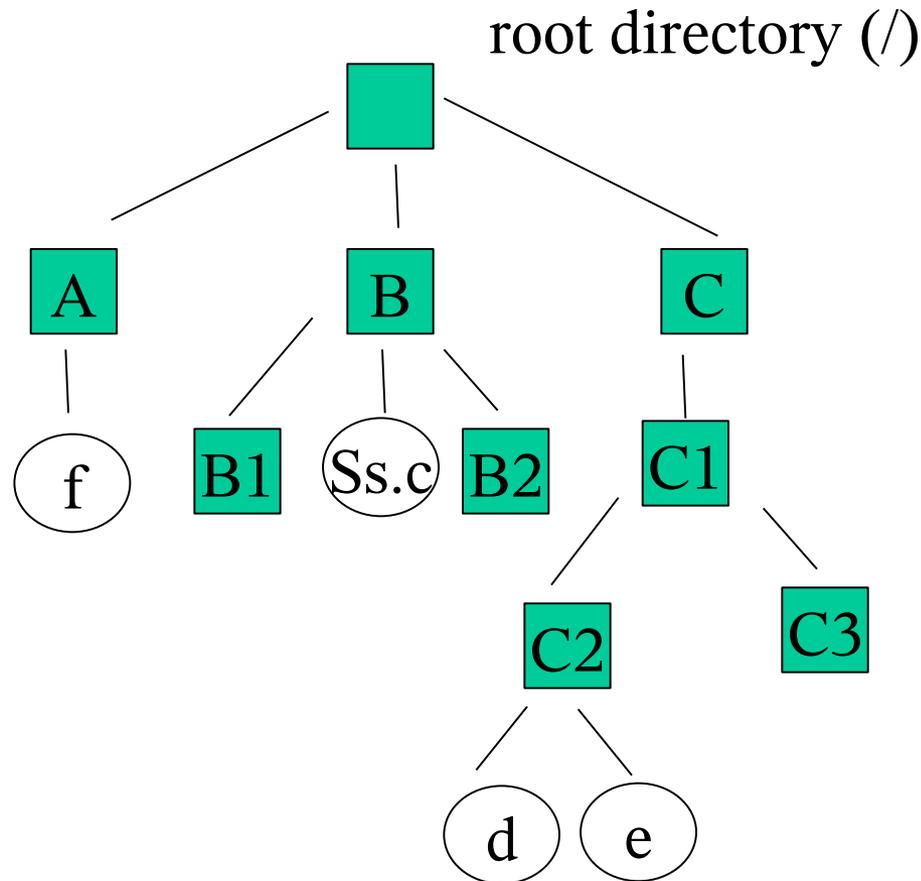
- Esempio di FS senza link file

# Path name assoluto



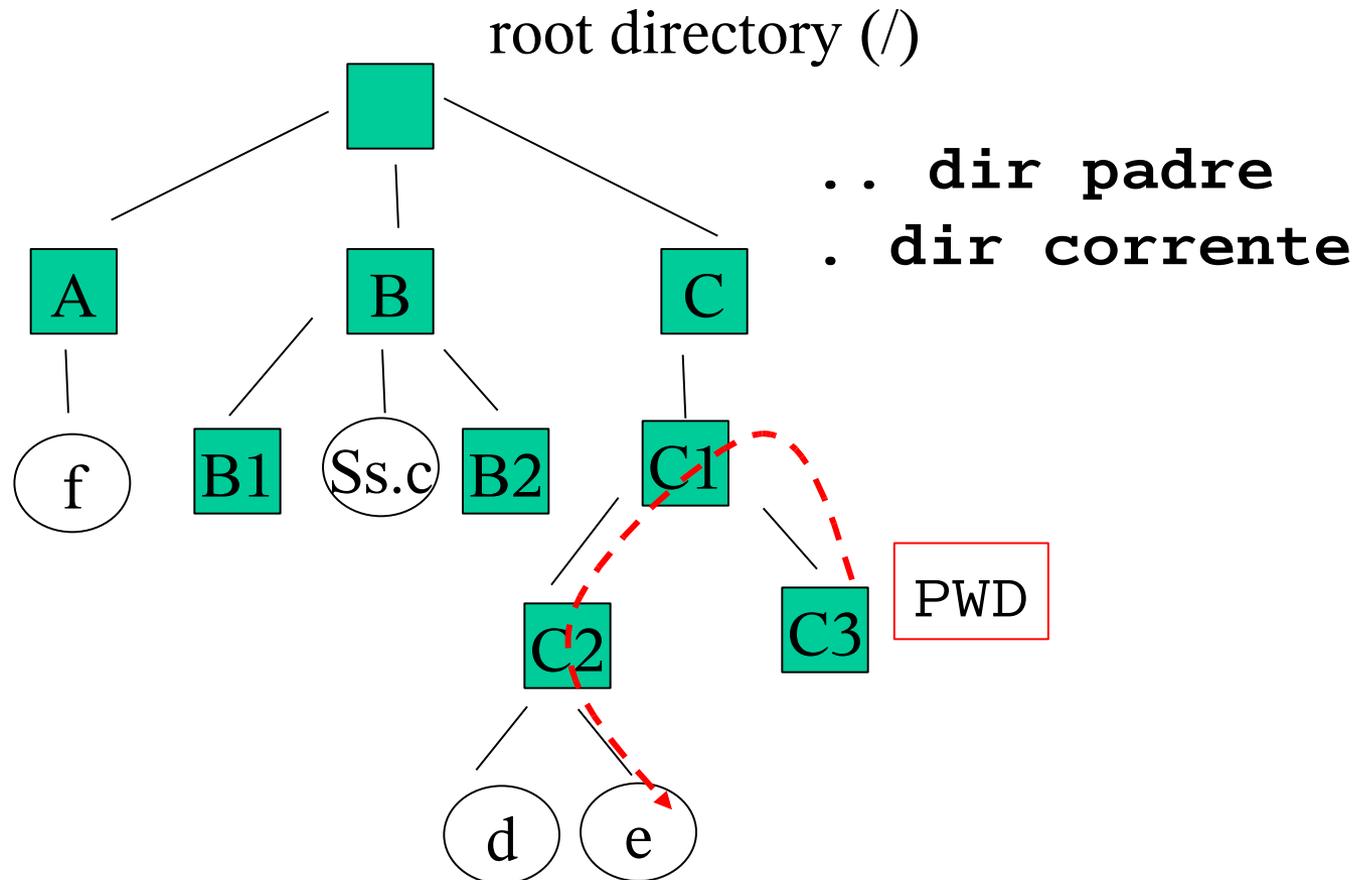
- Ogni file è univocamente determinato dal cammino che lo collega alla radice
  - /C/C1/C2/e

# Working directory e path name relativo



- Ogni processo Unix (anche la shell!) ha associata una *working directory*
  - si può vedere con **pwd** e si cambia con **cd**

# Path name relativo (2)



- Il PNR è il cammino dalla Working Directory
  - ../../C2/e

```
bash:~$ pwd
```

```
/home/s/susanna
```

```
bash:~$ cd PROVE
```

```
bash:~/PROVE$ pwd
```

```
/home/s/susanna/PROVE
```

```
bash:~/PROVE $ ls myFile
```

```
myFile
```

```
bash:~/PROVE$ ls ./myFile
```

```
-- equivalente a
```

```
-- ls Myfile
```

```
myFile
```

```
bash:~/PROVE$ cd ..
```

```
bash:~$ pwd
```

```
/home/s/susanna/
```

```
bash:~$
```

```
bash:~$ pwd
```

```
/home/s/susanna
```

```
bash:~$ cd PROVE
```

```
bash:~/PROVE$ pwd
```

```
/home/s/susanna/PROVE
```

```
-- file listing
```

Sta per /home/s/susanna

```
bash:~/PROVE $ ls myFile
```

```
myFile
```

```
bash:~/PROVE$ ls ./myFile -- equivalenti
```

```
myFile
```

```
bash:~/PROVE$ cd ..
```

```
bash:~$ pwd
```

```
/home/s/susanna/
```

```
bash:~$
```

```
bash:~$ pwd
```

```
/home/s/susanna
```

```
bash:~$ cd PROVE
```

```
bash:~/PROVE$ pwd
```

```
/home/s/susanna/PROVE
```

```
-- file listing
```

```
bash:~/PROVE $ ls myFile
```

```
myFile
```

```
bash:~/PROVE$ ls ./myFile -- equivalenti
```

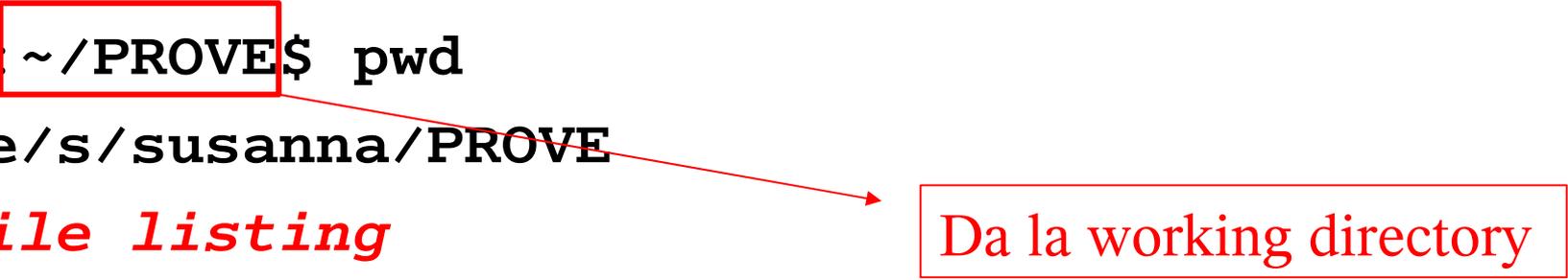
```
myFile
```

```
bash:~/PROVE$ cd ..
```

```
bash:~$ pwd
```

```
/home/s/susanna/
```

```
bash:~$
```



Da la working directory

*-- copiare un file :cp*

```
bash:~$ ls
```

```
a.out d.c d.h
```

```
bash:~$ cp d.c f.c
```

```
bash:~$ ls
```

```
a.out d.c d.h f.c
```

*-- differenze fra file?*

```
bash:~$ diff d.c f.c
```

*-- diff:nessun output == nessuna differenza*

*-- cp: 'i' chiede conferma prima di  
sovrascrivere*

```
bash:~$ cp -i d.c f.c
```

```
cp: overwrite 'f.c'? n
```

```
bash:~$
```

*-- ridenominare un file : mv*

```
bash:~$ ls
```

```
a.out d.c d.h f.c
```

```
bash:~$ mv f.c h.c
```

```
bash:~$ ls
```

```
a.out d.c d.h h.c
```

*-- mv: 'i' chiede conferma prima di  
sovrascrivere*

```
bash:~$ mv -i d.c h.c
```

```
mv: overwrite 'h.c'? n
```

```
bash:~$ ls
```

```
a.out d.c d.h h.c
```

```
bash:~$
```

*-- rimuovere un file : rm*

```
bash:~$ ls
```

```
a.out d.c d.h h.c
```

```
bash:~$ rm f.c h.c
```

```
rm: cannot lstat 'f.c': Nu such file or directory
```

```
bash:~$ ls
```

```
a.out d.c d.h
```

*-- rm: 'i' chiede conferma prima di sovrascrivere*

```
bash:~$ rm -i d.c
```

```
rm: remove regular file 'd.c'? n
```

```
bash:~$ ls
```

```
a.out d.c d.h
```

```
bash:~$
```

*-- ogni utente ha una HOME DIRECTORY denotata con '~username' oppure '~'*

```
bash:~/LCS$ cd ~susanna
```

```
bash:~$ pwd
```

```
/home/s/susanna
```

*--equivalente a 'cd' o cd ~*

```
bash:~/LCS$ cd ~
```

```
bash:~$
```

...

```
bash:~/LCS$ cd
```

```
bash:~$
```

*-- ls ha diverse opzioni*

*-- 'a' visualizza i file nascosti (.\*)*

```
bash:~$ ls -a
```

```
.bashrc      .bash_profile  .login
```

.....

*-- 'd' da info sulla directory*

```
bash:~$ ls -ld
```

```
drwxr-xr-x 6 susanna users ... .. Mar 3 2005 ./
```

*-- 'F' aggiunge un carattere per il tipo di file '/' per directory, '\*' per seguibile, '@' per link, '=' per socket e niente per i file ordinari*

```
bash:~$ ls -F
```

```
a.out* dd/ pippo.c
```

*-- creare directory : mkdir*

```
bash:~$ pwd
```

```
/home/s/susanna
```

```
bash:~$ mkdir TEMP
```

```
bash:~$ ls -F
```

```
a.out* dd/ pippo.c TEMP/
```

```
bash:~$
```

*-- rimuovere directory vuote : rmdir*

```
bash:~$ ls -F
```

```
a.out* dd/ pippo.c TEMP/
```

```
bash:~$ rmdir TEMP
```

```
bash:~$ ls -F
```

```
a.out* dd/ pippo.c
```

*-- rimuovere directory NON vuote : rm -r*

```
bash:~$ rmdir dd
```

```
rmdir: Directory not-empty
```

```
bash:~$
```

*-- rm: '-r' rimuove ricorsivamente la directory ed il suo contenuto, 'i' chiede conferma*

```
bash:~$ rm -ri dd/
```

```
rm: descend into directory 'dd'? n
```

```
bash:~$
```

*-- ridenominare directory : mv*

```
bash:~$ ls -F
```

```
a.out* dd/ pippo.c
```

```
bash:~$ mv dd TEMP
```

```
bash:~$ ls -F
```

```
a.out* pippo.c TEMP/
```

*-- modifica dei diritti*

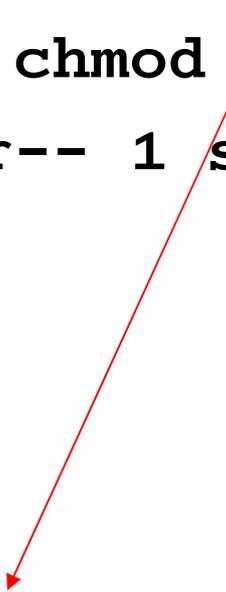
```
bash:~$ ls -l myfile
```

```
-rw-r--r-- 1 susanna user ... Feb 6 2005 myfile.c
```

```
bash:~$ chmod 664 myfile
```

```
-rw-rw-r-- 1 susanna user ... Feb 6 2005 myfile.c
```

```
bash:~$
```



Ottale!!

```
110 110 100
```

```
rw- rw- r--
```

*-- modifica i diritti*

```
bash:~$ ls -l myfile
```

```
-rw-r--r-- 1 susanna user ... Feb 6 2005 myfile.c
```

```
bash:~$ chmod 664 myfile
```

```
-rw-rw-r-- 1 susanna user ... Feb 6 2005 myfile.c
```

```
bash:~$ chmod ugo+x myfile
```

```
-rwxrwxr-x 1 susanna user ... Feb 6 2005 myfile.c
```

```
bash:~$
```

u- owner (susanna)  
g group (user)  
o - others  
a - all

+  
-  
=

r  
w  
x  
s

# Sul significato dei diritti

- File regolari
  - $r, w, x$  sui file regolari hanno un significato ovvio
- Directory:
  - $r$  su una directory significa avere il permesso di leggere i file appartenenti alla directory
    - es eseguire un **ls**
  - $w$  significa avere il permesso di modificare una directory :
    - es: cancellare aggiungere un file
  - $x$  significa aver il permesso di utilizzare una directory in un path (search permission)

# Sul significato dei diritti (2)

- File speciali (tipicamente in `/dev/...` )
  - *r* il processo può eseguire la system call read relativa al device corrispondente
    - es leggere da modem
  - *w* il processo può eseguire la system call write relativa al device corrispondente
    - es scrivere sulla stampante
  - *x* non ha nessun significato

# Regole di applicazione dei diritti

- Un processo in esecuzione ha associato 4 valori legati ai diritti:
  - *(real)user-ID*, *(real)group-ID* : intero positivo, associato al nome della login dell'utente che lo ha attivato ed al suo gruppo di appartenenza principale
    - vengono assegnati al momento dell'ingresso nel sistema
    - vengono ereditati da tutte le sottoshell e da tutti i processi creati da queste
  - *effective user-ID*, *effective group-ID* :

# Regole di applicazione dei diritti (2)

- In processo in esecuzione ha associato 4 valori legati ai diritti (cont.):
  - *effective user-ID, effective group-ID* : normalmente hanno lo stesso valore di *(real)user-ID, (real)group-ID*. In alcuni casi però differiscono
    - *es:* in situazioni in cui l'utente deve accedere a risorse normalmente protette (es cambiarsi la password richiede la scrittura del password file, riservata al superuser)

# Regole di applicazione dei diritti (3)

- se (*effective user-ID* == 0) il permesso è sempre accordato (*superuser*) altrimenti
- se (*effective user-ID* == *file\_userID*) vengono usati i primi tre bit altrimenti
- se (*effective group-ID* == *file\_groupID*) vengono usati i secondi tre bit altrimenti
- si usano gli ultimi tre bit

# Regole di applicazione dei diritti (4)

- Alcune note:
  - l'owner del file può avere meno diritti di tutti
  - il superutente ha sempre tutti i diritti
  - ci sono due permessi speciali '*set user-ID*' e '*set group-ID*' indicati da '*s*' invece di '*x*'
    - permettono di prendere effective userID e groupID da quelli del file eseguibile senza ereditarli da chi ci ha attivato
    - servono per eseguire cose come **passwd** senza essere superutente
  - cambiare owner a un file di solito non è permesso *...perché?*

*-- modifica owner*

```
bash:~$ chown miao myfile.c
```

```
-rw-r--r-- 1 miao user ... Feb 6 2005 myfile.c
```

*-- di solito solo la root può fare questo*

```
bash:~$
```

*-- modifica group*

```
bash:~$ chgrp miao myfile.c
```

```
-rw-r--r-- 1 susanna miao ... Feb 6 2005 myfile.c
```

*-- al cli non è permesso*

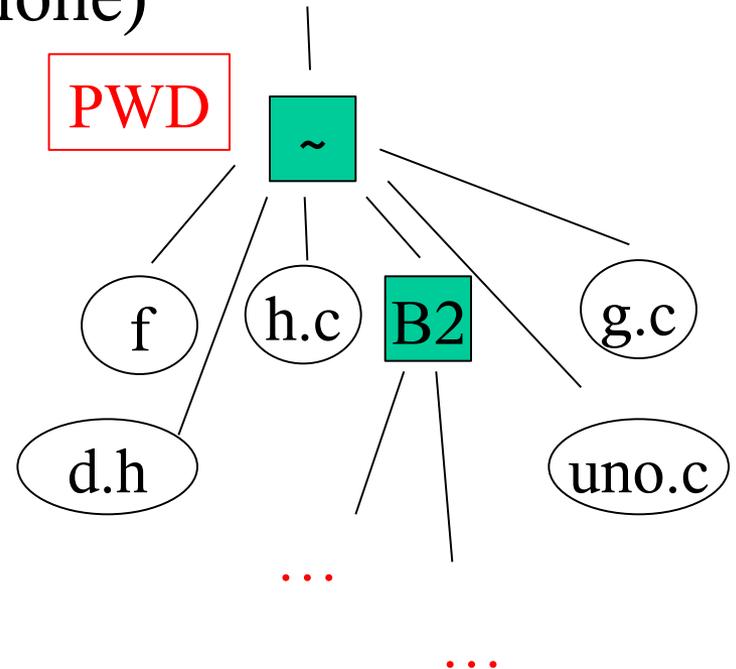
```
bash:~$
```

# Metacaratteri: *wildcard*

- *Wildcard* :
  - permettono di scrivere espressioni regolari che denotano un insieme di nomi di file
  - vengono risolti dalla shell durante l'*espansione di percorso* (prima dell'esecuzione)

– '\*' qualsiasi stringa

```
bash: ~$ ls *.c
```



# Metacaratteri: *wildcard* (2)

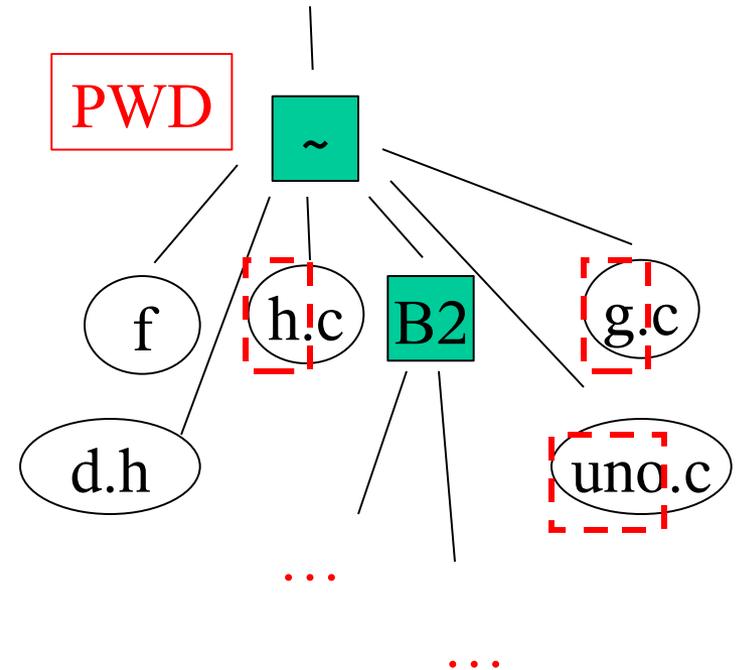
- *Wildcard* :

- ‘\*’ qualsiasi stringa

```
bash:~$ ls *.c
```

```
g.c h.c uno.c
```

```
bash:~$
```



# Metacaratteri: *wildcard* (3)

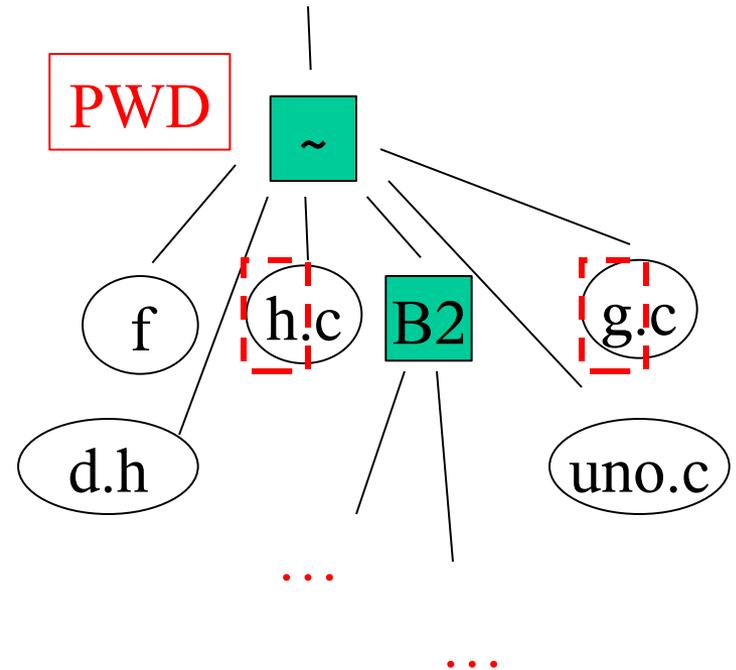
- *Wildcard* :

- ‘?’ qualsiasi carattere

**bash:~\$ ls ?.c**

**g.c h.c**

**bash:~\$**



# Metacaratteri: *wildcard* (4)

- *Wildcard* :

– ‘[...]’ insieme di caratteri

```
bash:~$ ls [ag].c
```

```
g.c
```

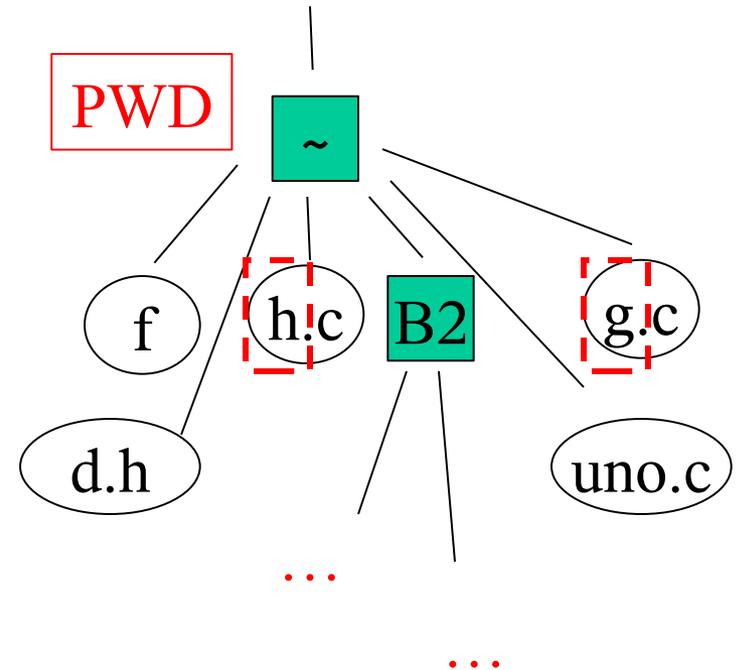
```
bash:~$ ls [!ag].c
```

```
h.c
```

```
bash:~$ ls [a-g].?
```

```
d.h g.c
```

```
bash:~$
```



# Metacaratteri: *wildcard* (5)

- Per capire meglio come funziona l'espansione di percorso usiamo il comando echo (visualizza la stringa argomento)

```
bash:~$ echo pippo
```

```
pippo
```

```
bash:~$ echo *.c
```

```
h.c g.c uno.c
```

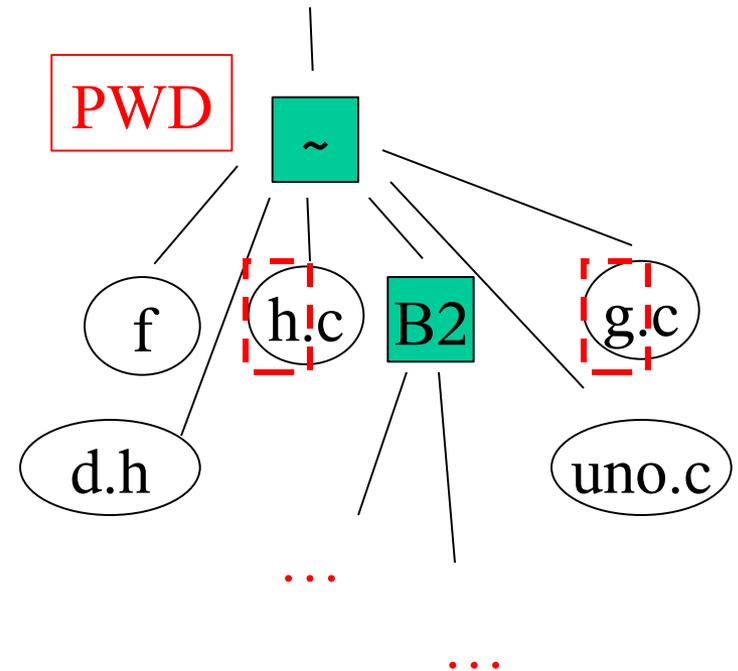
```
bash:~$ echo ?.c
```

```
g.c h.c
```

```
bash:~$ echo [h-z]*
```

```
h.c uno.c
```

```
bash:~$
```



# Visualizzare i file

**cat file1 ... fileN**

concatena il contenuto dei file e mostra tutto su stdout

**less file, more file**

permettono di navigare nel file, (vedi **man**)

**/<pattern>** cerca avanti

**?<pattern>** cerca indietro

**head [-n] file\_name**

mostra la prime 10 (o **n**) linee

**tail [-n] file\_name**

mostra la ultime 10 (o **n**) linee

# Cercare file/comandi: **find**

```
find <path> -name <fname> -print
```

dove

- **<path>** indica la directory da cui iniziare la ricerca. La ricerca continuerà in ogni sottodirectory.
- **<fname>** è il nome del file da cercare (anche un pattern costruito con metacaratteri)
- **-print** mostra i risultati della ricerca
- e molto altro (vedi **man**)

- **esempio:**

```
bash:~$ find . -name nn* -print
```

- cerca i file che iniziano per 'nn' nella directory corrente

# Cercare file/comandi: **locate**

- **find** è molto pesante
- **locate <pattern>**
  - cerca i file usando un database periodicamente aggiornato (con **updatedb**) ed è molto più efficiente

– esempi:

```
bash:~$ locate basen
```

```
/usr/bin/basename
```

```
.....
```

```
/usr/share/man/man1/basename.1.gz
```

```
/usr/share/man/man3/basename.3.gz
```

```
bash:~$
```

# Cercare programmi: **whereis**

- **whereis [-bms] <command>**
  - cerca la locazione di un programma fra i binari, i sorgenti o le pagine di manuale
  - [-b] binari, [-m] manuali e [-s] sorgenti es:

```
bash:~$ whereis -b eclipse
```

```
eclipse: /usr/local/eclipse
```

```
bash:~$ whereis -sm eclipse
```

```
eclipse:
```

```
bash:~$ whereis -b emacs
```

```
emacs: /usr/bin/emacs /etc/emacs
```

```
  /usr/lib/emacs /usr/share/emacs
```

```
bash:~$
```

# Cercare programmi: **which**

- **which <command>**
  - serve per capire quale copia di un comando sarà eseguita (pathname) fra quelle disponibili
  - esempi:  
**bash:~\$ which emacs**  
**/usr/bin/emacs**  
**bash:~\$**

# Cercare programmi: **type**

- **type [-all -path] <command>**
  - comando interno (builtin) di Bash simile a **which** ma più completo
    - indica come la shell interpreta **command**, specificandone la natura (alias, funzione, file eseguibile, builtin, parola chiave della shell)

– esempi:

```
bash:~$ type -all rm
rm is aliased to 'rm -i'
rm is /bin/rm
bash:~$ type -all for
for is a shell keyword
bash:~$
```

# Gestire archivi: **tar**

**tar [-ctvx] [-f file.tar] [<file/dir>]**

- permette di archiviare parti del filesystem in un unico file, mantenendo le informazioni sulla gerarchia delle directory

- c crea un archivio

- t mostra il contenuto di un archivio

- x estrae da un archivio

- f **file.tar** specifica il nome del file risultante

- v fornisce informazioni durante l'esecuzione

- esempi:

```
bash:~$ tar cf log.tar mylogs/log10*
```

```
bash:~$
```

# Gestire archivi: tar (2)

*-- guardare il contenuto*

```
bash:~$ tar tf log.tar
```

```
mylogs/log10_1
```

```
mylogs/log10_2
```

```
mylogs/log10_3
```

```
bash:~$
```

*-- estrarre sovrascrivendo i vecchi file*

```
bash:~$ tar xf log.tar
```

*-- l'opzione -k impedisce di sovrascrivere file  
con lo stesso nome*

```
bash:~$ tar xkf log.tar
```

```
bash:~$
```

# Comprimere file: **gzip** **bzip2**

```
gzip [opt] file
```

```
gunzip [opt] file.gz
```

```
bzip2 [opt] file
```

```
bunzip2 [opt] file.bz2
```

- permette di ridurre le dimensioni dei file con algoritmi di compressione della codifica del testo (lossless) , **gzip** (Lempel-Ziv coding LZ77) e **bzip2** (Burrows-Wheeler block sorting text compression algorithm)

- esempi:

```
bash:~$ gzip log*
```

```
bash:~$ gunzip relazione.doc.gz
```

```
bash:~$
```

# Filtrare i file: **grep** etc.

```
grep [opt] <pattern> [ file(s) ... ]
```

– *Get Regular Expression and Print*

- cerca nei file specificati le linee che hanno contengono il pattern specificato e le stampa sullo standard output

– esempi:

```
bash:~$ grep MAX *.c *.h
```

```
mymacro.h: #define MAX 200
```

```
rand.h: #define MAX_MIN 4
```

```
bash:~$ grep Warn *.log
```

```
sec3.log: LaTeX Warning: There were undefined  
references
```

```
bash:~$
```

# Filtrare i file: **grep** etc (2)

*-- '-i' case insensitive*

```
bash:~$ grep -i MAX mymacro.h
```

```
#define MAX 200
```

```
#define Max_two 4
```

```
bash:~$
```

*-- '-v' prints all lines that don't match the pattern*

```
bash:~$ grep -v MAX mymacro.h
```

```
#define MIN 1
```

```
#define Max_two 4
```

```
.....
```

```
bash:~$
```

# C'è molto di più ...

- Ma non facciamo in tempo a parlarne
  - sed, awk, Perl ...
- Un semplice esempio.

*-- tr translate or delete characters*

```
bash:~$ more g.txt
```

```
prova
```

```
prova
```

```
bash:~$ tr " " "b" < g.txt
```

```
prova
```

```
bbbbbbbbprova
```

```
bash:~$
```

# C'è molto di più ... (2)

- Faremo una carrellata più avanti dei tool per la gestione di progetti C complessi
  - **make**, e altro ...