Introduzione alla shell di Linux: bash (**b**ourne **a**gain **sh**ell)

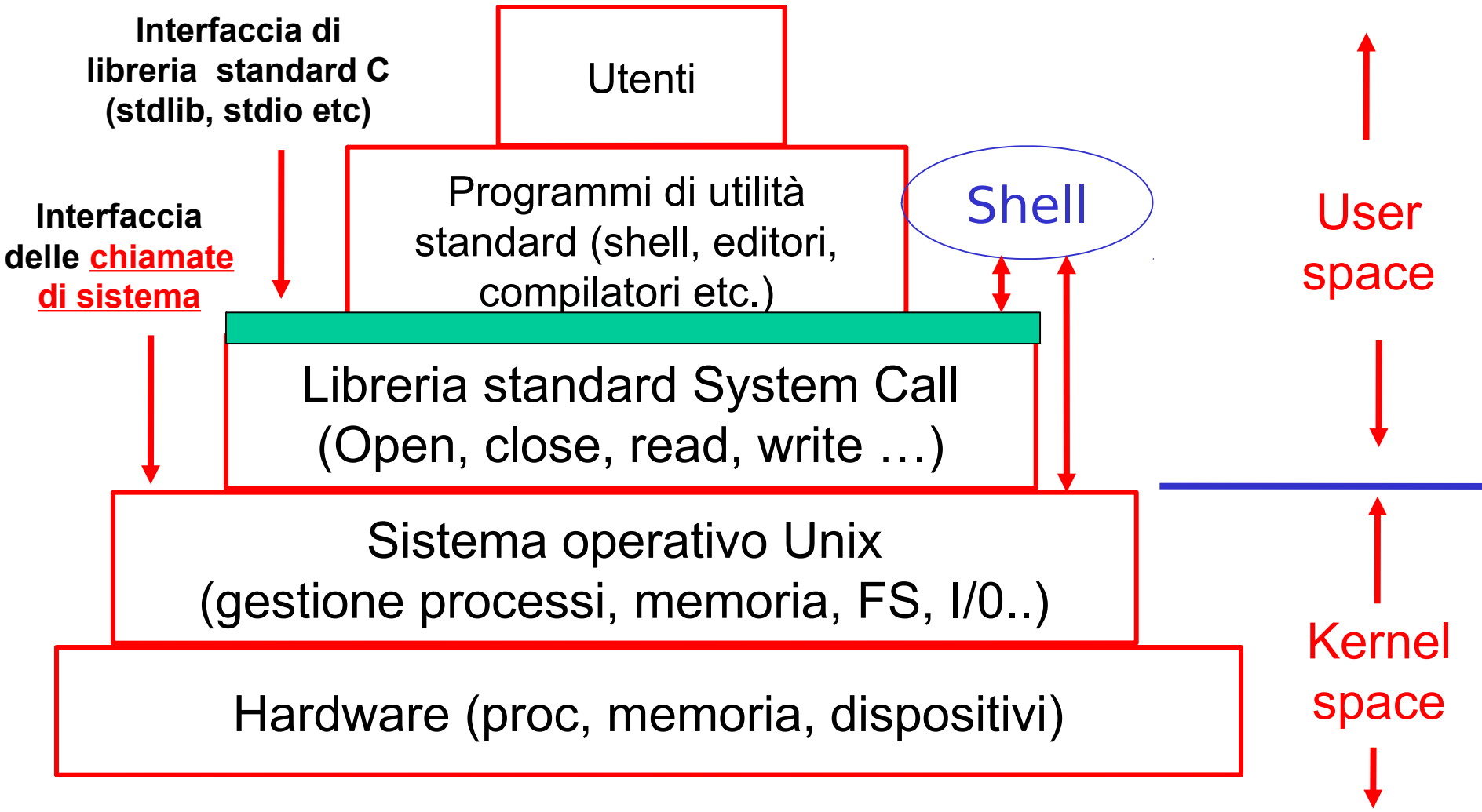
La filosofia della shell Unix

- Si rivolge a programmatori
- Mette a disposizione comandi con sintassi minimale
- 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`** ***--lo vedremo più avanti***

La filosofia della shell Unix (2)

- È ancora oggi il principale programma di interfaccia
  - la shell è un normale programma senza privilegi speciali
  - gira in spazio utente
  - interfaccia testuale
- È usata spesso come supporto per automatizzare attività di routine
  - shell scripting
  - configurazioni di sistema

UNIX/Linux: shell



Cos'è una shell .....

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

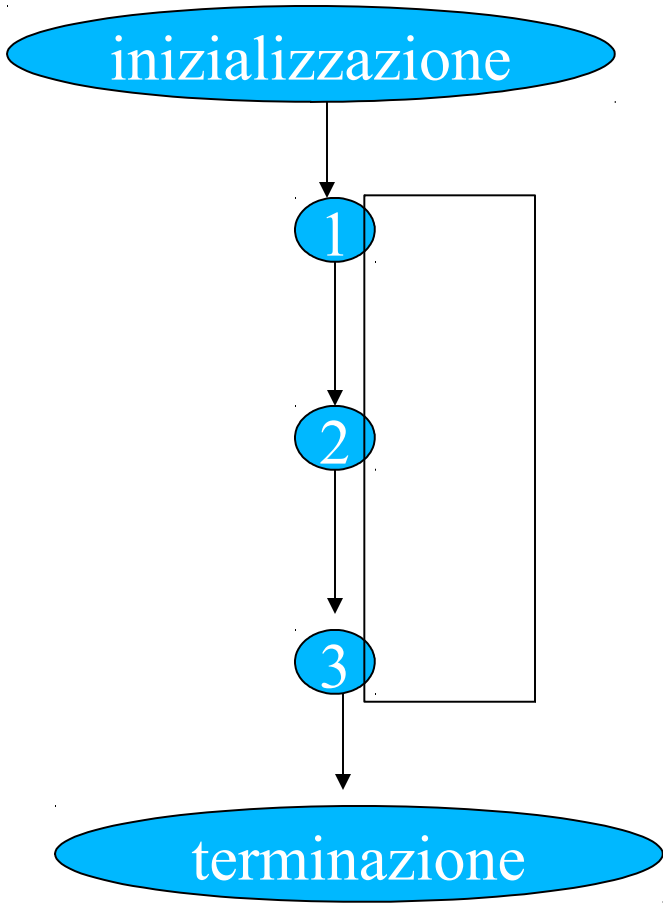
```
bash:~$                    -- (prompt)nuovo comando?
bash:~$ date               -- l'utente da il comando
Thu Mar 12 10:34:50 CET 2005  -- esecuzione
bash:~$                    -- (prompt) nuovo comando?
```

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*

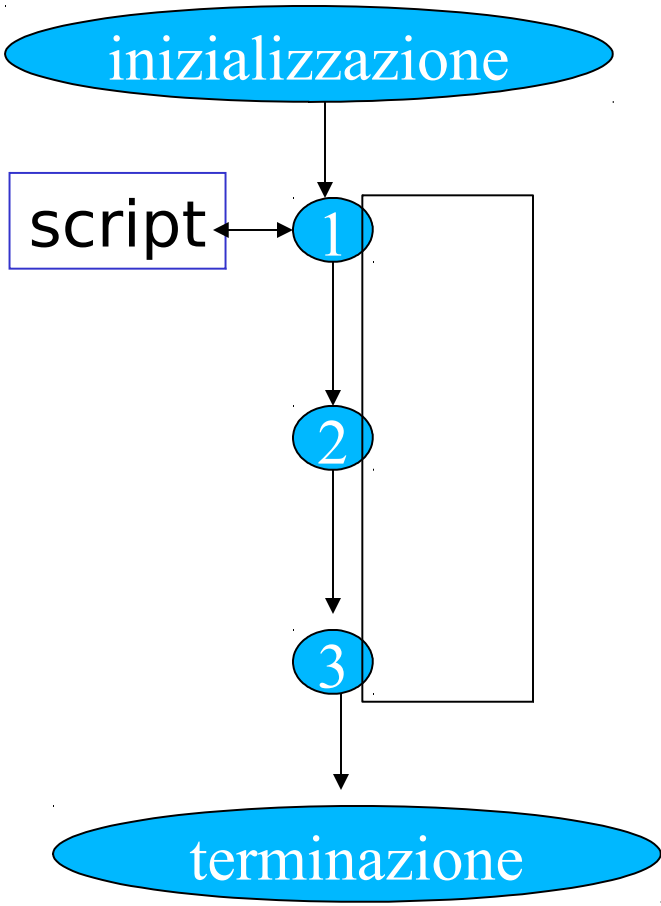


Cos'è una shell ..... (3)

- 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*



Cos'è una shell ..... (4)

- Ci sono vari tipi di shell
  - C shell (**`csh,tcsh`**), Bourne shell (**`sh`**), Bourne Again shell (**`bash`**), Debian Alquist Shell (**`dash, sh`** Debian recenti)
- 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 tipo IF, WHILE etc.
    - più altro...

Comandi base di Unix

- Sintassi tipica:

```
comando <opzioni> <argomenti>
```

- es:

```
-- trovare la data
bash:~$ date                    --no options
                                --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
```

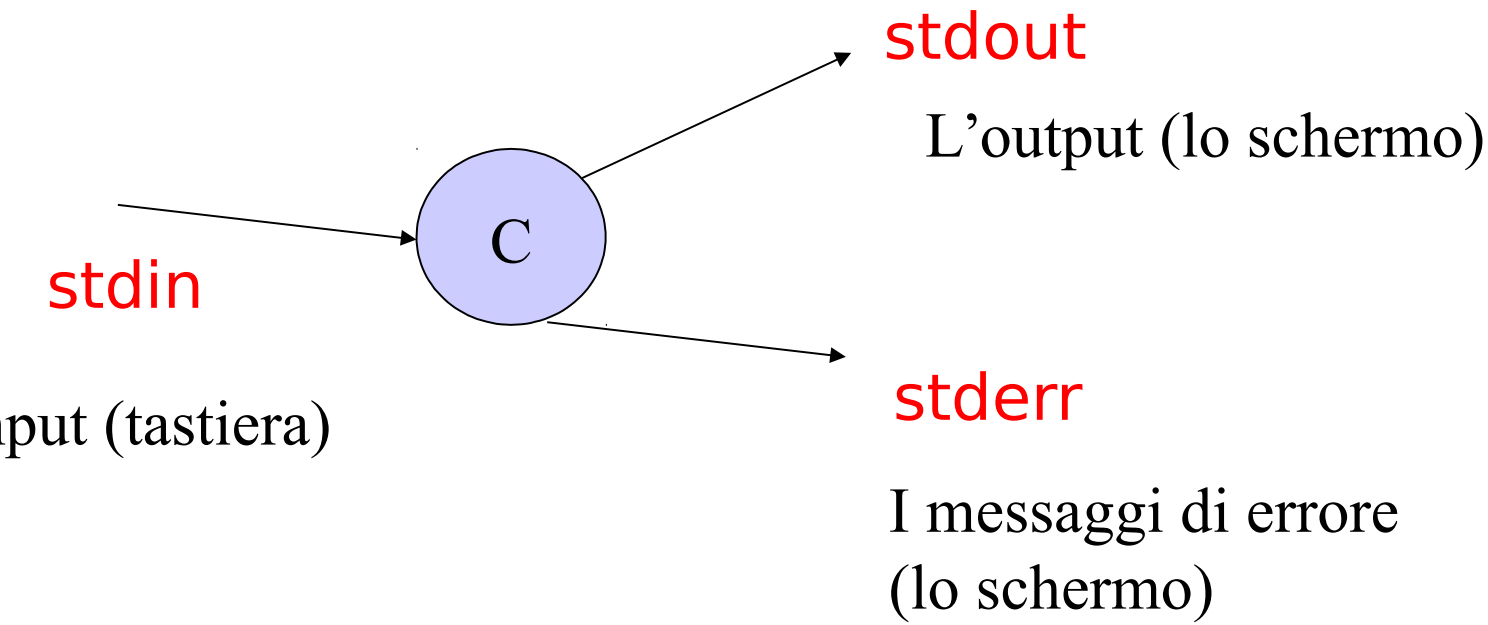
bash:~$

Standard input, ouput ed error

- Negli esempi visti l'output viene scritto sempre su terminale
- Ogni comando o programma che gira sotto Unix ha sempre tre stream di I/O attivi:
  - **`stdin`** lo standard input
  - **`stdout`** lo standard ouput
  - **`stderr`** lo standard error
- Di default questi tre streams sono collegati al terminale di controllo del processo che sta eseguendo il programma o il comando di shell
  - meccanismo semplice e flessibile
  - Ridirezione (più avanti)

Standard input, ouput ed error (2)

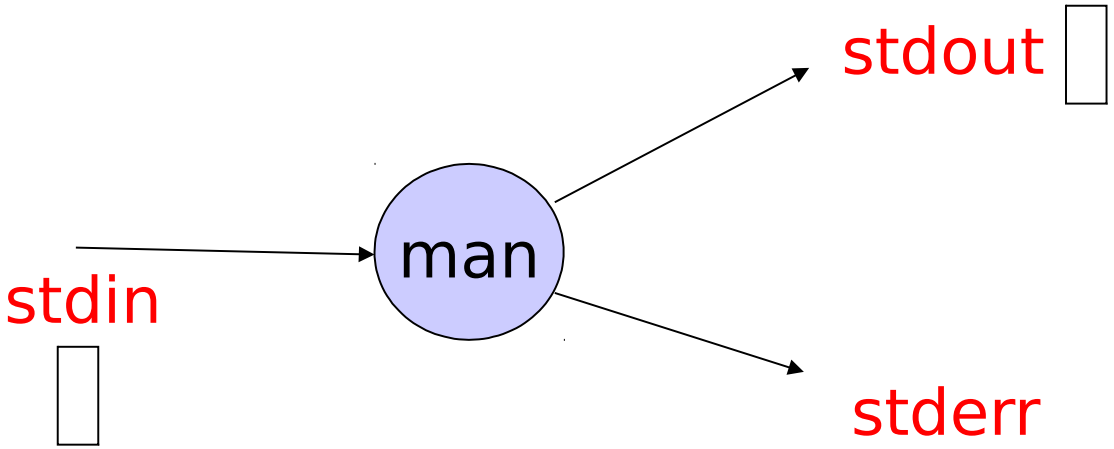
- Tipicamente



Standard input, ouput ed error (3)

– Es.

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



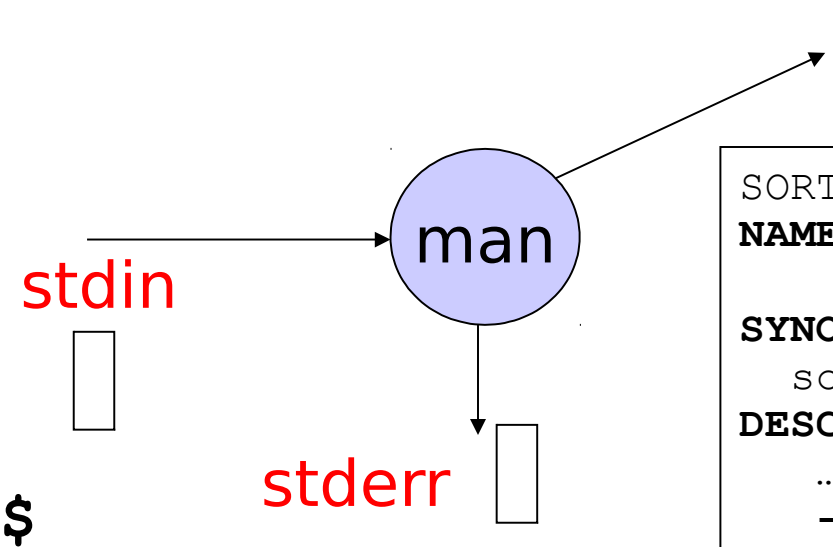

```
bash:~$
```

```
No manual entry for ciccio
```

Standard input, ouput ed error (4)

– Es.

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




```
bash:~$
```

```
SORT(1)    User Commands   …
NAME
     sort - sort lines ….
SYNOPSIS
   sort [ OPTION ] … [ FILE ] …
DESCRIPTION
    …
    -f --ignore-case
        fold lower case to ...
AUTHOR … REPORTING BUGS….
SEE ALSO
    Textinfo    info sort
```

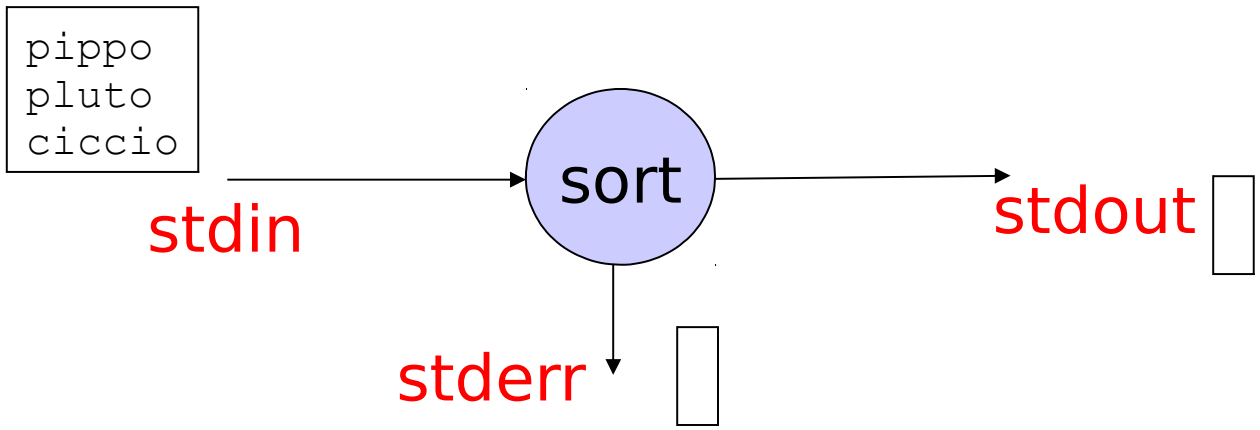
Standard input, ouput ed error (5)

```
bash:~$ sort
```

-- attende input dall'utente

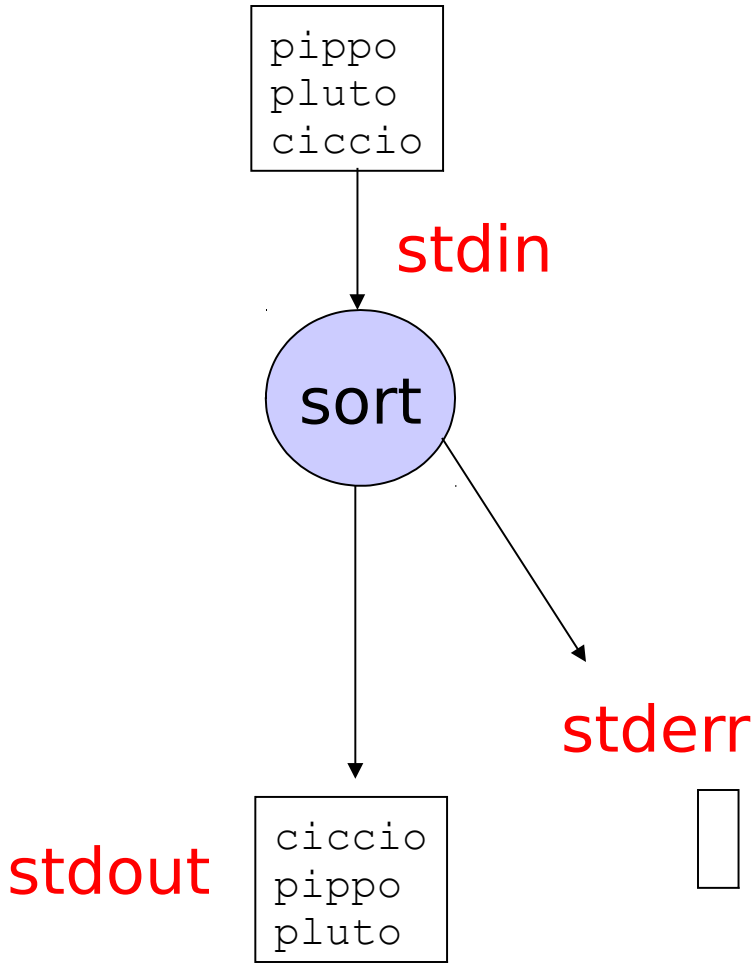
Standard input, ouput ed error (6)

```
bash:~$ sort
pippo
pluto
ciccio
```



Standard input, ouput ed 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(s)

```
bash:~$ ls --help
usage: ls [OPTION] [FILE]
List information about files ……
-a --all         do not ignore entries starting with .
-A --almost-all do not list . and ..
-b --escape      print octal escapes for non graphic
            characters
…
…

bash:~$
```

Standard command options

- Esistono un insieme di linee guida per scrivere utility standard per i sistemi Unix
  - **`http://www.gnu.org/prep/standards`**
  - i comandi Unix comuni si uniformano a queste direttive, e lo stesso dovrebbero fare gli script che scriviamo
- Si prevedono due tipi di opzioni:
  - corte (un solo carattere con `-`) come: **`-a -l -u`**
  - lunghe (più caratteri con `--`) come:

`--help --version --all`

  - entrambe possono richiedere un parametro

`-o file --output=file`

Standard command options (2)

- Ci sono due opzioni che dovrebbero essere sempre fornite

`--help`

  - stampa sullo standard output una breve documentazione del programma, dove inviare gli eventuali **`bug`** e termina con successo

`--version`

  - stampa sullo standard output nome del programma, versione, stato legale ed terminare con successo.

- Ci sono utility che permettono di effettuare agevolmente il parsing di opzioni con questo formato

-- Comandi più lunghi di una riga

-- echoing

```
bash:~$ echo This is a very long shell command \
and needs to be extended with the \
  line continuation character…
This is a very long shell command and needs to be
extended with the line continuation character…
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

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

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 /bin/ls
ls: ELF 32-bit LSB executable Intel 80386
version 1 (SYSV), GNU/Linux 2.2.0,dynamically
linked (uses shared libs), stripped
bash:~$ file pippo.n
pippo.n: ASCII text
bash:~$
```

Esempi: **cut**

```
-- selezione parti di linee in un file
bash:~$ cut -d":" -f 1  /etc/passwd
root
deamon
bin
...
-- tipo di file
bash:~$ cut -c 2- /etc/passwd
oot:x:0:0:root:/root:/bin/bash
aemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
in:x:2:2:bin:/bin:/usr/sbin/nologin
...
```

Editing della linea di comando

- Molte shell, fra cui la bash, offrono funzioni di *editing della linea di comando* "ereditate" da un editor. Nel nostro caso è 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 (↑↓) ….. troppo lungo
- … con CTRL-r posso navigare iniziando a digitare un comando

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
-- la shell c TAB eta
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)

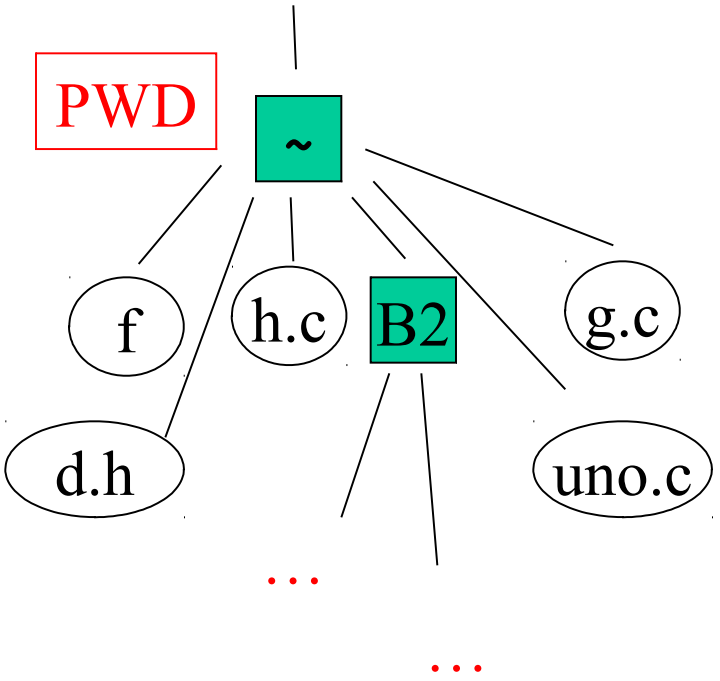
- se esistono più completamenti possibili
  - premendo TAB viene emesso un segnale sonoro.
  - Premendolo nuovamente si ottiene la lista di tutti i file che iniziano con il prefisso già digitato.

```
bash:~$ ls
un_file  uno.c ns2.h
bash:~$ cp un
un_file  uno.c
bash:~$ cp un
```

TAB TAB

Metacaratteri: *wildcard*

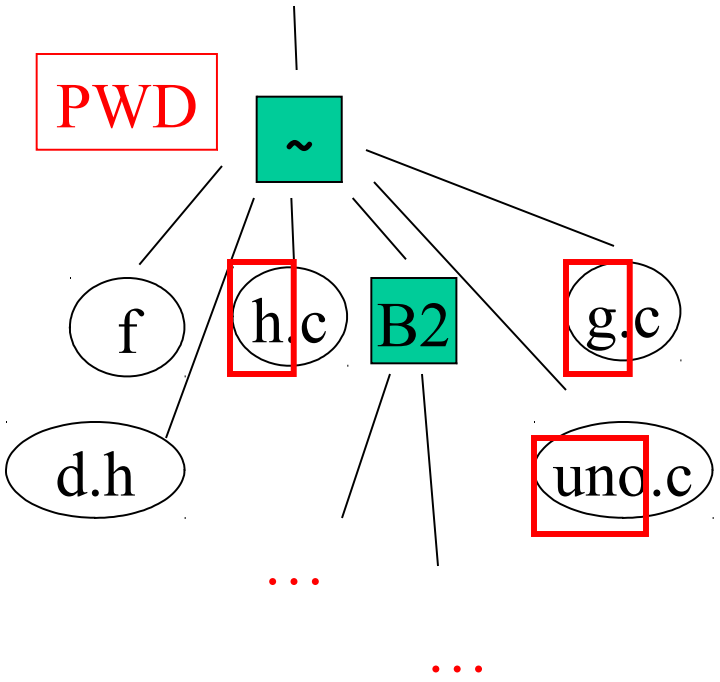
- *Wildcard* :
  - permettono di scrivere pattern che denotano un insieme di stringhe (*wildcard expansion* or *globbing*)
  - vengono usate dalla shell durante l'*espansione di percorso* sui nomi di file (prima dell'esecuzione) oppure in altri costrutti di shell che prevedono pattern matching (es **`case`**)
- i principali sono 2
  - '?' qualsiasi carattere
  - '*' qualsiasi stringa
    eventualmente vuota



Metacaratteri: *wildcard* (2)

- *Wildcard* :
  - '*' qualsiasi stringa

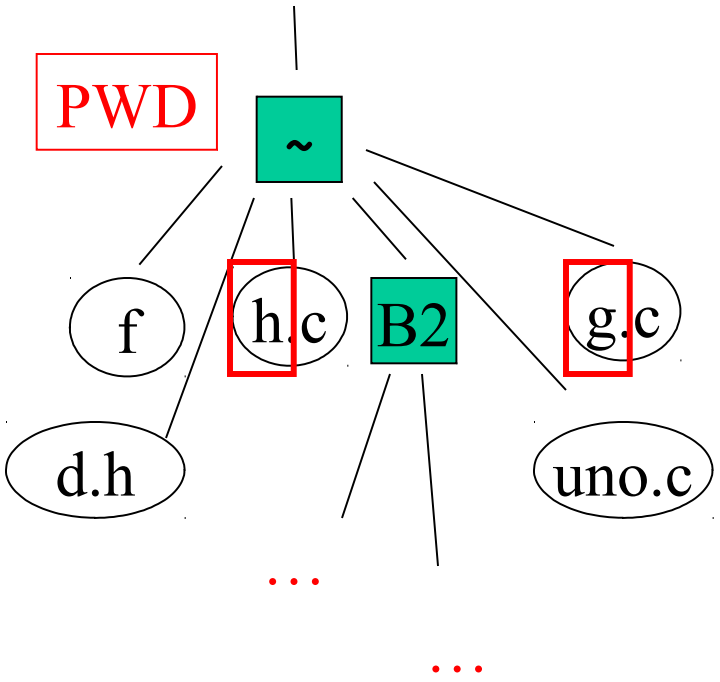
```
bash:~$ ls *.c
g.c h.c uno.c
bash:~$
```



Metacaratteri: *wildcard* (3)

- *Wildcard* :
  - '?' qualsiasi carattere

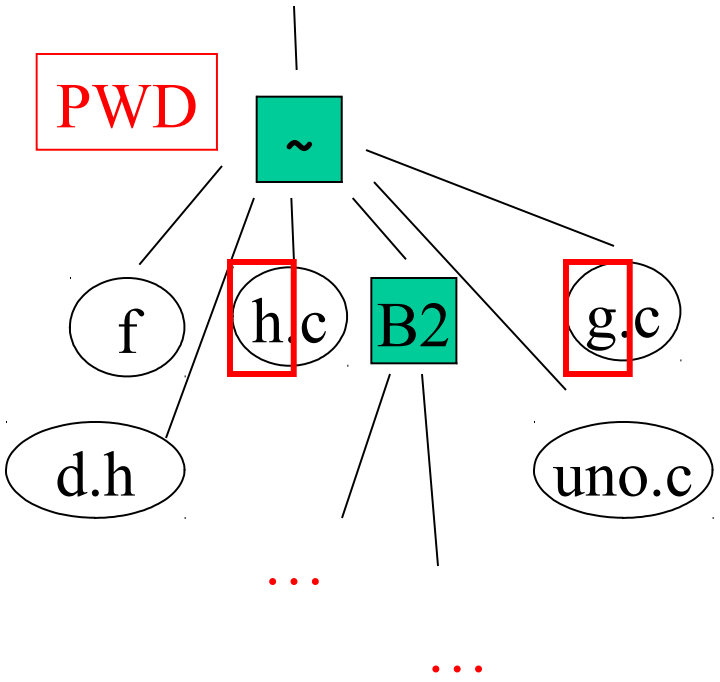
```
bash:~$ ls ?.c
g.c h.c
bash:~$
```



Globbing esteso

- Altri costrutti per esprimere i pattern
  - '[…]' insieme di caratteri (funziona solo nell'espansione di percorso, non **`case`**)

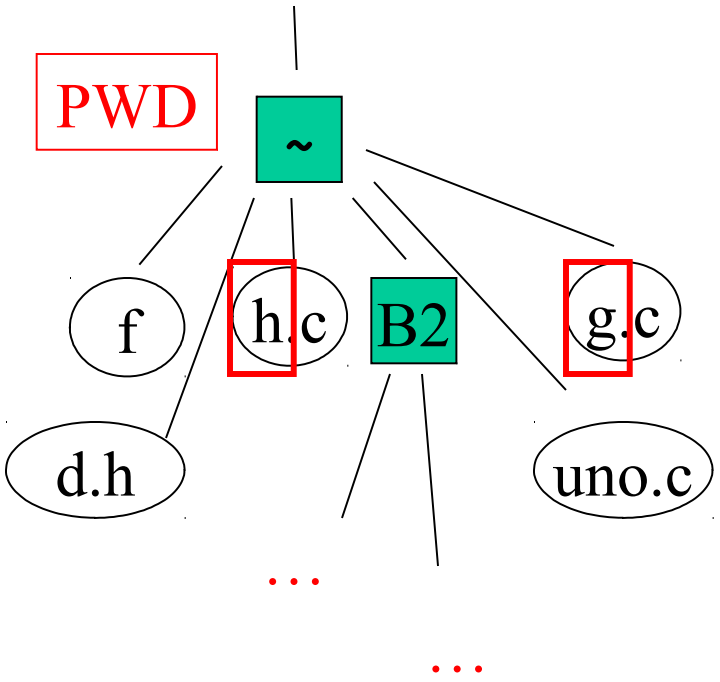
```
bash:~$ ls [ag].c
g.c
bash:~$ ls [!ag].c
h.c
bash:~$ ls [a-g].?
d.h g.c
bash:~$
```



Ancora globbing

- 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:~$
```



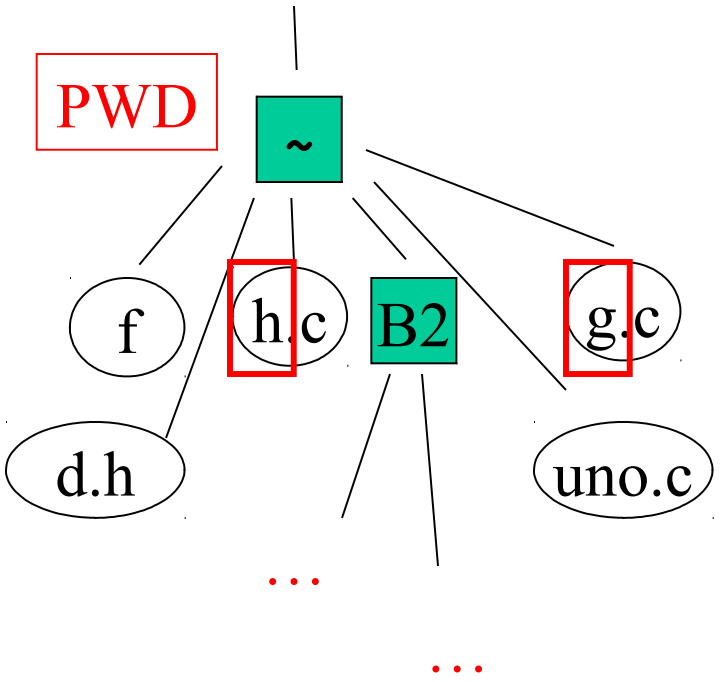
Ancora globbing (2)

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

```
bash:~$ echo [hi].c
h.c
```

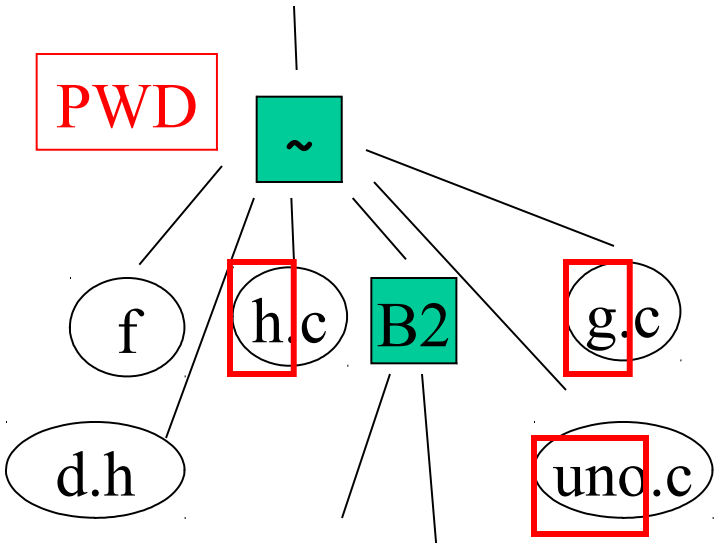
da non confondere con la espansione delle graffe!

```
bash:~$ echo {h,i}.c
h.c i.c
```



Globbing: se non c'è match

- *Globbing*: se non c'è matching
  - viene comunque restituito il pattern con wildcard, es:




```
bash:~$ echo *.f
*.f
bash:~$ echo ?.f
?.f
bash:~$ ls [ab].g
ls: cannot access [ab].g: No such file or
directory
```

Alcuni comandi....

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: `find (2)`

- Esempio più complesso:

```
bash:~$ find . -name "*.[ch]" -exec cp {} {}.bkp \; -print
```

  - cerca tutti i file .c e .h a partire dalle directory corrente e per ognuno di essi crea un copia con estensione .bkp

Cercare file/comandi: **locate**

- **find** può essere "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 tvf log.tar
mylogs/log10_1
mylogs/log10_2
mylogs/log10_3
bash:~$
-- estrarre sovrascrivendo i vecchi file
bash:~$ tar xvf log.tar
-- l'opzione -k impedisce di sovrascrivere file
con lo stesso nome
bash:~$ tar xvkf log.tar
bash:~$
```

Comprimere file: **gzip bzip2**

```
gzip [opt] file
gunzip [opt] file.gz
bzip2 [opt] file
bunzip2 [opt] file.bz2
```

- permette di ridure 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**

grep [opt] <pattern> [file(s) …]

- *Get Regular Expression and Print*
  - cerca nei file specificati le linee che 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** (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:~$
```

Filtrare i file: **tr**

```
tr
-- tr translate or delete characters
bash:~$ more g.txt
prova
        prova
bash:~$ tr " " "b" < g.txt
prova
bbbbbbbbprova
bash:~$
```

C’è molto di più ...

- **`sed, awk, ...`**
- ma non li tratteremo
- Alcuni esempi:
- - dos2unix usando sed:
- **`sed -i 's/\r//' files*`**
- - numerare tutte le righe dei file in input:
- **`awk '{print FNR "\t" $0}' files*`**