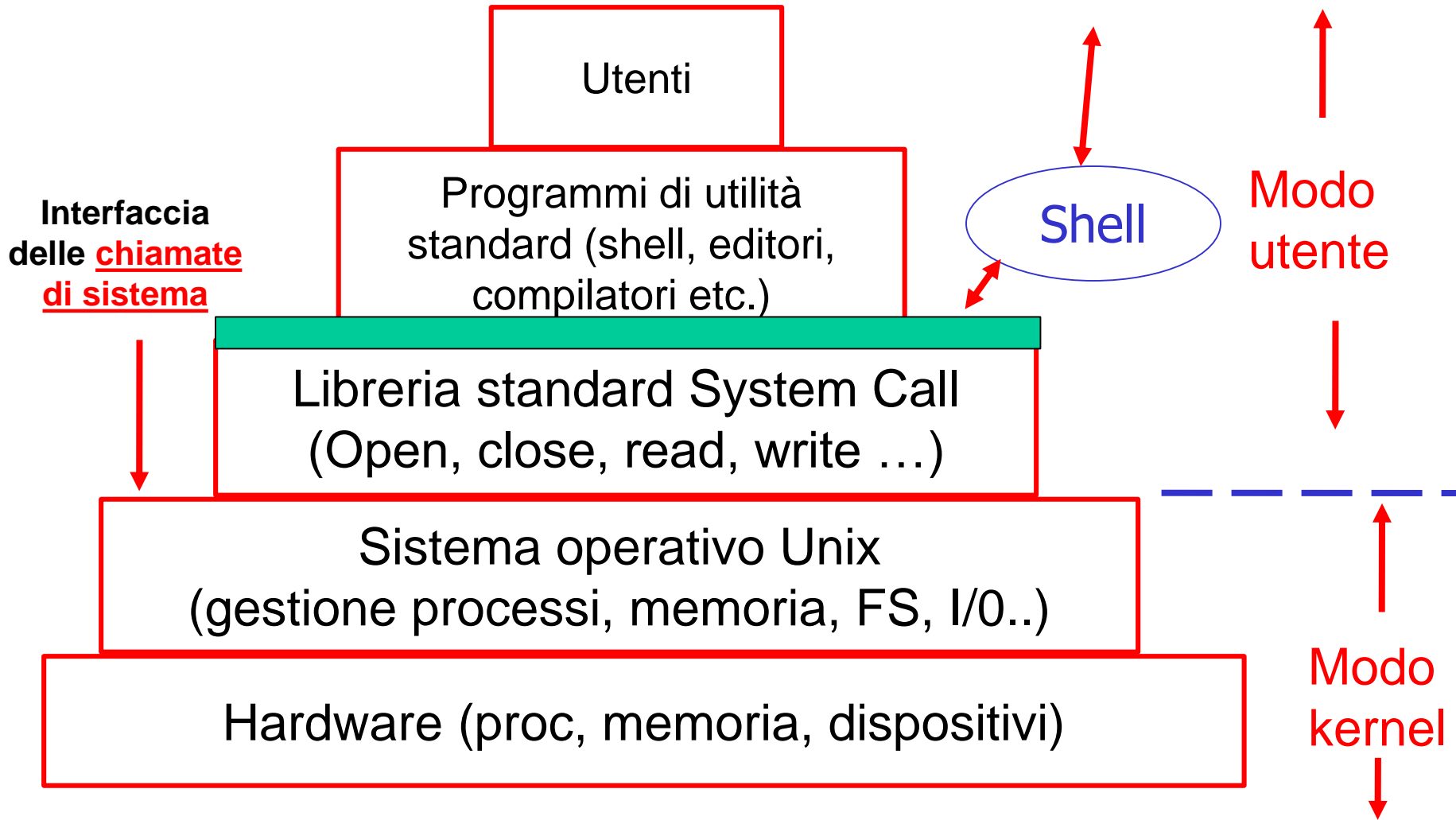


# Cenni di Bash

# Obiettivi

- Assumiamo la conoscenza di base di bash
  - Interazione e comandi principali
  - Alias, history
  - Wild cards (\*, ?)
  - Pipelining e ridirezione (|,>,<,>>)
- Approfondiremo la struttura interna della shell:
  - variabili, espansione della riga di comando, comandi composti (liste, pipe, sequenze condizionali)
- Daremo le basi di programmazione di shell (*shell scripting*)
  - Funzioni, costrutti di controllo, debugging

# UNIX/Linux: shell



# 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?*

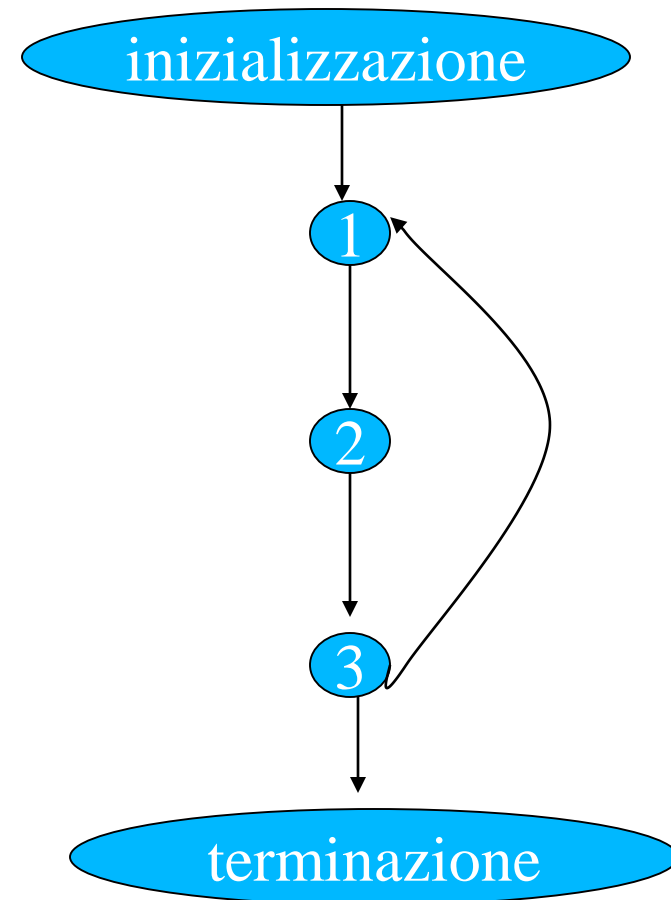
`bash:~$ date`                *-- l'utente da il comando*

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

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

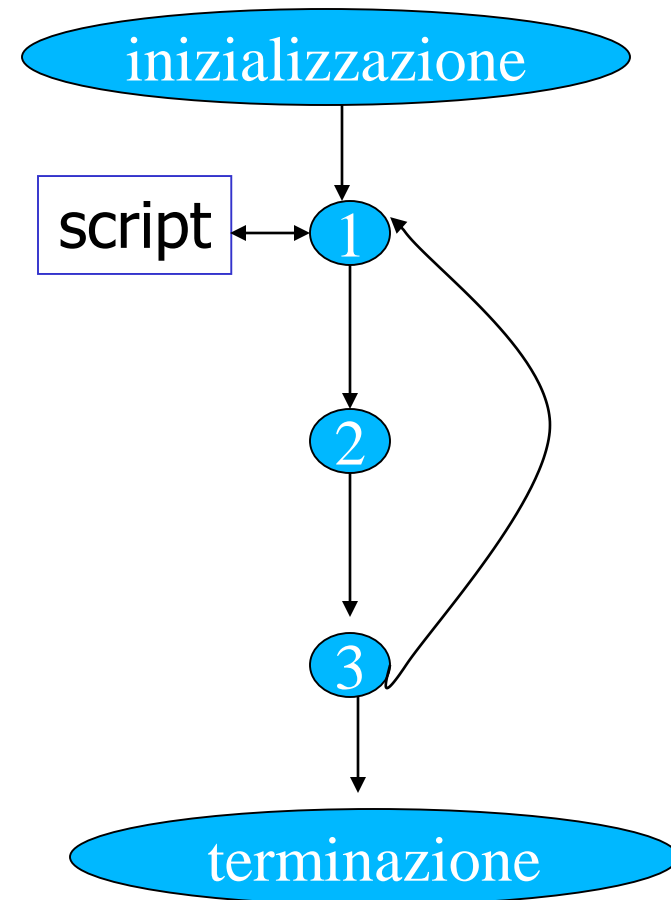
# Cos'è una shell ..... (2)

- 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*



# Shell scripting

# Un primo esempio di script

```
bash$ cat prova.sh  
echo "Hallo World !"  
bash$
```



# Un primo esempio di script

- Come procedere per l'esecuzione di uno script:
  - salvare i comandi sopra in un file (**prova.sh**)
    - attenti al separatore (newline)
  - assicurarsi che su **prova.sh** sia permessa l'esecuzione
  - lanciare **la bash** con lo script (ed i suoi eventuali argomenti) come argomento

# Un primo esempio di script

```
bash:~$ bash prova.sh
```

```
Hallo Wold!
```

```
bash:~$
```

# Uno script con argomenti

```
bash$ cat prova.sh
echo "Script $0"
echo "Primo Parametro $1"
echo "Secondo Parametro $2"
bash$
```

# Uno script con argomenti

```
bash:~$ bash ./prova.sh ciccio pippo
```

```
Script ./prova.sh
```

```
Primo Parametro ciccio
```

```
Secondo Parametro pippo
```

```
bash:~$
```

# #!/bin/bash

- In realtà possiamo specificare la shell direttamente nello script

```
bash$ cat prova1.sh
```

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

```
echo "Script $0"
```

```
echo "Primo Parametro $1"
```

```
echo "Secondo Parametro $2"
```

```
bash:~$ ls -l prova1.sh
```

```
-rwxr-xr-x 1 susanna ... Feb 6 2016 prova1.sh
```

```
bash$
```

# # ! /bin/bash

- Il risultato è lo stesso di prima, ma non è necessario invocare la bash esplicitamente

```
bash$ ./prova1.sh gg ff dd
Script ./prova1
Primo Parametro gg
Secondo Parametro ff
bash$
```

- Questo è quello che faremo in tutti i nostri script

# Variabili di shell

- Le variabili della shell:
  - una variabile è un *nome* cui è associato un *valore*
    - nome: stringa alfanumerica che comincia per lettera
    - valore: stringa di caratteri
  - per dichiarare/assegnare un valore ad una variabile  
**<varname>=[<value>]**
    - se **varname** non esiste viene creata altrimenti il valore precedente viene sovrascritto
    - attenzione: prima e dopo il segno '=' non devono comparire spazi

# Variabili di shell

- Una variabile si dice *definita* quando contiene un valore
  - anche la stringa vuota!
- Può essere cancellata con  
`unset varname`
- Per riferire il valore si usa la notazione  
`$<varname>` oppure `${<varname>}`



# Variabili di shell

- Script con variabili:

```
bash$ cat prova.sh
#!/bin/bash
RADIX=pippo
FILE2=pluto.c
# stampa pippo.h e pippo.c
cat ${RADIX}.h ${RADIX}.c
# stampa pluto.c
cat $FILE2
bash$
```

# Variabili di shell predefinite

- Alcuni variabili sono assegnate da Bash, es:

**SHELL**            *-- shell di login*

**HOSTTYPE**        *-- tipo di host, es i386-linux*

**HISTSIZE**        *-- numero cmd nella history*

**HISTFILE**        *-- file dove salvare la history*

– Per vederle tutte : **set**

- esempi:

```
bash$ echo $HISTSIZE
```

```
500
```

```
bash$ echo $HISTFILE
```

```
/home/s/susanna/.bash_history
```

```
bash$
```

# Variabili di shell: PS1

- Controllare il prompt:

- **PS1** controlla il *prompt primario*, quello della shell interattiva. Alcune stringhe hanno un significato particolare

- **\u** nome dell'utente
- **\s** nome della shell
- **\v** versione della shell
- **\w** working directory
- **\h** hostname

- esempio:

```
bash$ PS1= '\u@\h: \w$'
```

```
susanna@fujih1:~$ PS1= '\s$'
```

```
bash$
```

# Variabili di shell: **PATH**

- *Search path*: alcune variabili sono legate ai path dove cercare comandi e directory
  - **PATH** serie di directory in cui viene cercato il comando da eseguire, es:  

```
bash$ echo $PATH
```

```
/usr/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11:
```

```
bash$
```
  - normalmente è predefinita

# Variabili di shell: **PATH** (2)

- *Eseguire comandi nella directory corrente*

```
bash:~$ echo $PATH
```

```
/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11
```

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

```
myscript*
```

```
bash:~$ myscript
```

```
bash: myscript: command not found
```

```
bash:~$ ./myscript
```

```
Hallo World!
```

```
bash:~$
```

# Variabili di ambiente

- Le variabili di shell fanno parte dell'ambiente *locale* della shell stessa
  - quindi non sono visibili a processi o sottoshell attivate
  - una classe speciale di variabili, dette variabili di ambiente, sono invece visibili anche ai sottoprocessi
  - una qualsiasi variabile può essere resa una **variabile d'ambiente** esportandola:

```
export <varnames> --esporta
```

```
export <varname>=<value> --defin e esporta
```

```
export --lista variabili esportate
```

# Variabili di ambiente (2)

- Alcune variabili locali sono esportate di default:
  - es **HOME**, **PATH**, **PWD**
  - le definizioni in **.bashrc** sono valide in ogni shell interattiva

- **Esempi:**

```
bash:~$ PATH=$PATH:.
```

```
bash:~$ bash -- creo una sottoshell
```

```
bash:~$ echo $PATH
```

```
/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11:.
```

```
bash:~$ -- PATH è stata ereditata
```

# Parametri speciali (alcuni)

- \$0** Nome dello script
- \$\*** Insieme di tutti i parametri posizionali a partire dal primo. Tra apici doppi rappresenta un'unica parola composta dal contenuto dei parametri posizionali .
- \$@** Insieme di tutti i parametri posizionali a partire dal primo. Tra apici doppi rappresenta una serie di parole, ognuna composta dal contenuto del rispettivo parametro posizionale.  
Quindi "\$@" equivale a "\$1" "\$2" "\$3" ...
- \$\$** PID (*process identifier*) della shell



# Parametri speciali (alcuni) (2)

- Esempio

```
bash$ more scriptArg.sh
```

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

```
echo Sono lo script $0
```

```
echo Mi sono stati passati $# argomenti
```

```
echo Eccoli: $*
```

```
bash$ ./scriptArg.sh ll kk
```

```
Sono lo script ./scriptArg
```

```
Mi sono stati passati 2 argomenti
```

```
Eccoli: ll kk
```

```
bash$
```

# Controllo del flusso

If, while etc...

# Strutture di controllo

- Permettono di
  - condizionare l'esecuzione di porzioni di codice al verificarsi di certi eventi
  - eseguire ripetutamente alcune parti etc.
- Bash fornisce tutte le strutture di controllo tipiche dei programmi imperativi
  - vengono usate soprattutto negli script ma si possono usare anche nella linea di comando

# Strutture di controllo (2)

## – **if-then-else**

- esegue una lista di comandi se una condizione è / non è vera

## – **for**

- ripete una lista di comandi un numero prefissato di volte

## – **while, until**

- ripete una lista di comandi finchè una certa condizione è vera / falsa

## – **case**

- esegue una lista di comandi scelta in base al valore di una variabile

## – **select**

- permette all'utente di scegliere fra una lista di opzioni

# Costrutto **if**

- esegue liste di comandi differenti, in funzione di condizioni espresse anch'esse da liste di comandi
- sintassi (usando ‘;’ come terminatore della condizione)

```
if <condition>; then  
    <command-list>  
[elif <condition>; then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

# Costrutto **if** (2)

– sintassi (usando ‘newline’ come terminatore)

```
if <condition>  
then  
    <command-list>  
[elif <condition>  
then  
    <command-list>] ...  
[else  
    <command-list>]  
fi
```

# Costrutto **if** (3)

- Semantica:

- esegue la lista di comandi `<condition>` che segue `if`
- se l'*exit status* è 0 (*vero*) esegue `<command-list>` che segue `then` e termina
- altrimenti esegue le condizioni degli `elif` in sequenza fino a trovarne una verificata
- se nessuna condizione è verificata esegue la `<command-list>` che segue `else`, se esiste, e termina
- l'*exit status* è quello dell'ultimo comando eseguito (0 se non ha eseguito niente)

# Comandi e builtin

- Un comando richiesto alla shell può
  - corrispondere a un *file eseguibile* localizzato da qualche parte nel file system (si parla di *comando*) oppure
  - può corrispondere ad una funzionalità implementata internamente alla shell (si parla di *builtin*)



# Comando

- Ad esempio:

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

```
bash:~$ ls
```

- il file eseguibile viene ricercato in tutte le directory contenute nella variabile di ambiente PATH
- se il file esiste : la shell crea un nuovo processo shell (usando la SC **fork**) che cura l'esecuzione del programma contenuto nel file eseguibile (utilizzando la SC **exec**). La shell padre si mette in attesa della terminazione del figlio (SC **waitpid**) e poi ristampa il prompt

# Builtin

- la shell esegue direttamente il builtin al suo interno senza attivare altri processi

– es, cambio della working directory

```
bash:~$ cd
```

```
bash:~$
```

– es, scrittura di una stringa su stdout

```
bash:~$ echo ciao
```

```
ciao
```

```
bash:~$
```

# Costrutto **if** (4)

- **Usò tipico**

- siccome 0 significa esecuzione non anomala:

```
if <esecuzione regolare del comando>; then  
    <elaborazione normale>  
else  
    <gestione errore>  
fi
```

# Costrutto **if** : esempi

- Esempio: eseguiamo cd e poi ls

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

```
if cd $1; then
```

```
    echo "$0: File listing:"
```

```
    ls
```

```
else
```

```
    echo "$0: Error" 1>&2
```

```
fi
```

# Costrutto **if** : esempi (3)

- Esempio: eseguiamo cd e poi ls

```
bash:~$ ./ifscript .  
./ifscript: File listing:  
pippo.c a.out mio.txt  
bash:~$ ./ifscript gigi  
./ifscript: Error  
bash:~$
```

# Condizione: combinare *exit status*

- **&&**, **||**, **!** (and, or, negazione) si possono usare per combinare gli exit status nelle condizioni
- Es: verifichiamo che un file contenga una di due parole date:

```
file=$1; wrd1=$2; wrd2=$3;
if grep $wrd1 $file || grep $wrd2 $file; then
    echo "$wrd1 o $wrd2 sono in $file"
fi
```

- analogamente se ci sono entrambe ...

# Test

- La condizione dell'**if** è un comando (possibilmente composto) ma questo non significa che si può testare solo la terminazione di un comando
- con la seguente sintassi

`test <condition> oppure [ <condition> ]`

- si può controllare:
  - proprietà dei file (presenza, assenza, permessi...)
  - confronti tra stringhe e interi
  - combinazioni logiche di condizioni

# Test - stringhe

- Alcuni confronti fra stringhe:
  - con la condizione di verità

<code>str1 = str2</code>	<i>se str1 e str2 sono uguali</i>
<code>str1 != str2</code>	<i>se str1 e str2 sono diverse</i>
<code>-n str1</code>	<i>se str1 non è nulla</i>
<code>-z str1</code>	<i>se str1 è nulla</i>



# Costrutto **if** : esempi (2)

- Esempio: eseguiamo `cd` e poi `ls`

```
#!/bin/bash
cd $1;
if [ $? = 0 ]; then
    echo "$0: File listing:"
    ls
else
    echo "$0: Error" 1>&2
fi
```

# Test - attributi file

- `-e file` *se file esiste*
- `-d file` *se file esiste ed è directory*
- `-f file` *se file esiste e non è speciale  
(dir, dev)*
- `-s file` *se file esiste e non è vuoto*
- `-x -r -w file` *controlla diritti  
esecuzione, lettura e scrittura*
- `-O file` *se sei l'owner del file*
- `-G file` *se un tuo gruppo è gruppo di file*
- `file1 -nt file2`
- `file1 -ot file2`  
*se file1 è più nuovo (vecchio) di file2  
(data ultima modifica)*

# Costrutto **if** : esempi (3)

- Esempio: inseriamo controlli nel precedente script

```
#!/bin/bash
if [ $# = 0 ]; then
    echo "Usage: $0 dirname" 1>&2
elif ! [ -d $1 ]; then
    echo "$0 : $1: Not a directory" 1>&2
elif cd $1; then
    echo "$0: File listing:"
    ls
else
    echo "$0: Error cannot cd to $1" 1>&2
fi
```

# Costrutto **for**

- Permette di eseguire un blocco di istruzioni un numero prefissato di volte. Una variabile, detta *variabile di loop*, assume un valore diverso ad ogni iterazione
- diversamente dai costrutti **for** dei linguaggi convenzionali non permette di specificare *quante* iterazioni fare, ma una *lista di valori assunti dalla variabile di loop*. Sintassi

```
for <var> [ in <list> ]; do  
  <command-list>  
done
```

- se **<list>** è omessa si assume la lista degli argomenti dello script (**\$@**)

# Costrutto `for` (2)

- Semantica:
  - Espande l'elenco `<list>` generando la lista degli elementi
  - Eseguce una scansione degli elementi nella lista (separatore il primo carattere in `$IFS`)
  - Alla variabile `<var>` ad ogni iterazione viene assegnato un nuovo elemento della lista e quindi si esegue il blocco `<command-list>` (che tipicamente riferisce la variabile di loop)
  - L'*exit status* è quello dell'ultimo comando eseguito all'interno della lista `do` oppure 0 se nessun comando è stato eseguito

# Costrutto **for** : esempi

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

```
#Applica cat a tutti gli argomenti
```

```
if [ $# = 0 ]; then
```

```
    echo "Usage: $0 file1 ... fileN" 1>&2
```

```
fi
```

```
for FILE in $@ ; do
```

```
    cat $FILE
```

```
done
```

# Costrutto **for** : esempi (2)

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

```
#Applica cat a tutti gli argomenti
```

```
if [ $# = 0 ]; then
```

```
    echo "Usage: $0 file1 ... fileN" 1>&2
```

```
fi
```

```
#non importa specificare la lista
```

```
for FILE do
```

```
    cat $FILE
```

```
done
```

# Costrutto **for** : esempio C-like

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

```
#stampa i numeri pari fino a 20
```

```
for ((i=0; i<=20; i+=2)); do
```

```
    echo $i
```

```
done
```



# Costrutto **for** : esempio C-like (2)

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

```
#calcola i numeri di Fibonacci minori di 200
```

```
echo Ecco i numeri di Fibonacci ...
```

```
for (( i=1, j=1; j<=200; k=i, i=j, j=i+k ))
```

```
do
```

```
    echo $j
```

```
done
```

# Costrutto **case**

- Permette di confrontare una stringa con una lista di pattern, e di eseguire di conseguenza diversi blocchi di istruzioni (simile a switch in C, Java)
- Sintassi:

```
case <expr> in  
    <pattern>  
        <command-list> ;;  
    <pattern>  
        <command-list> ;;  
...  
esac
```

# Costrutto case (2)

– Sintassi alternativa:

```
case <expr> in
  (<pattern>)
    <command-list> ;;
  (<pattern>)
    <command-list> ;;
...
esac
```

# Costrutto case (2)

- Semantica:

- L'espressione **<expr>** (in genere una variabile) viene espansa e poi confrontata con ognuno dei **<pattern>**
  - stesse regole dell'espansione di percorso (?,\*)
  - il confronto avviene in sequenza
- Se un pattern viene verificato si esegue la lista di comandi corrispondente e si esce
- Ogni pattern può in realtà essere l'or di più pattern  
**<pattern1> | ... | <patternN>**
- L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito

# Costrutto **case** : esempio

- Lo script

**mycd dir file.tar**

- che con 1 o 0 parametri stampa la variabile **\$PWD**
- mentre con 2 parametri copia in **dir**, il file **file.tar** e lo decompone (con **tar**)
- con più di 2 parametri da errore

# Costrutto **case** : esempio (2)

```
#!/bin/bash
case "$#" in
    ( 0|1 ) echo $PWD;;
    ( 2 ) if [ -d -x -w $1 ]; then
            cp $2 $1
            cd $1
            tar xvf $2
        fi ;;
    ( * ) echo "$0: too many args" 1>&2 ;;
esac
```

# Costrutto `select`

- Permette di generare un menu e gestire la scelta da tastiera dell'utente

- Sintassi:

```
select <var> [ in <list> ]; do
```

```
  <command-list>
```

```
done
```

- **Semantica:**

- il comando `<list>` viene espanso generando una lista di elementi (se è assente si usa "\$@" )

# Costrutto `select` (2)

- **Semantica (cont):**
  - ogni elemento della lista viene proposto sullo standard error (ognuno preceduto da un numero) .
    - Quindi viene mostrato il prompt di `$PS3` (di default `$`) e chiesto il numero della scelta all'utente
  - la scelta fatta viene memorizzata nella variabile **REPLY** e l'elemento corrispondente della lista in `<var>`
  - con una scelta non valida il menu viene riproposto
  - se è valida si esegue `<command-list>` e si ripete tutto
  - si esce con il builtin **break**
  - L'exit status è quello dell'ultimo comando eseguito oppure 0 se nessun comando è stato eseguito



# Costrutto `select` : esempio

- Lo script

**`icd`**

- che elenca le directory presenti in quella corrente
- e a scelta dell'utente si sposta in una di queste ed effettua il listing dei file presenti

# Costrutto `select`: esempio (2)

```
#!/bin/bash
PS3="Scelta?"
select dest in $(command ls -aF | grep "/"); do
    if [ -d -x -r $dest ]; then
        cd $dest;
        echo "icd: Changed to $dest"
        ls
        break
    else
        echo "icd: wrong choice" 1>&2
    fi
done
```

# Costrutti **while** **until**

- Permettono di ripetere l'esecuzione di un blocco di istruzioni fino al verificarsi (**while**) o al falsificarsi (**until**) di una condizione

- Sintassi:

```
while <condition>; do           until <condition>; do  
  <command-list>                <command-list>  
done                             done
```

- **<condition>** è analogo a quello dell'**if**
- al solito vera (0), falsa (!=0)
- L'exit status è quello dell'ultimo comando di **<command-list>** oppure 0 se non si entra nel ciclo

# Consigli per il debugging ...

# Prima di tutto ...

- **ATTENZIONE:**
  - gli script possono essere pericolosi, proteggete file e directory ed eseguiteli in ambienti non danneggiabili finchè non siete ragionevolmente sicuri della loro correttezza!
  - Attenzione a lasciare gli spazi dove servono ed agli effetti delle espansioni!

# Opzioni per il debugging

- Alcune opzioni utili per il debugging:
  - settabili con comando **set** [-/+o]
  - **noexec -n** : non esegue, verifica solo la correttezza sintattica
  - **verbose -v** : stampa ogni comando prima di eseguirlo

# Opzioni per il debugging (2)

- Alcune opzioni utili per il debugging (cont):

- **xtrace -x** : mostra il risultato dell'espansione prima di eseguire il comando

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

```
pippo.c pluto.c
```

```
bash:~$ set -x
```

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

```
+ ls -F pippo.c pluto.c
```

```
pippo.c pluto.c
```

```
bash:~$
```

# Funzioni



# Funzioni

- Bash offre la possibilità di definire *funzioni*
  - un funzione associa un *nome* ad un *programma di shell* che viene mantenuto in memoria e che può essere richiamato come un comando interno (builtin)

```
[function] <nome> () {  
    <lista di comandi>  
}
```
- Le funzioni sono eseguite nella shell corrente
  - e non in una sottoshell come gli script

# Funzioni (2)

- Parametri posizionali e speciali sono utilizzabili come negli script
  - es. possono essere usate per definire alias con parametri

```
rmall () {  
    find . -name "$1" -exec rm -i {} \; ;  
}
```
- Le funzioni si possono cancellare con

```
unset -f funct_name
```

# Funzioni (3)

- Per vedere le funzioni definite in fase di inizializzazione della shell ...

```
bash:~$ declare -f
```

fornisce tutte le funzioni ed il loro codice sullo standard output

```
bash:~$ declare -F
```

fornisce i nomi di tutte le funzioni (senza il codice)

```
bash:~$ type -all name_function
```

fornisce tutte le informazioni ed il codice della funzione di nome

```
name_function
```

- Vediamo alcuni esempi ....

# Funzioni (4)

```
bash:~$ rmall () { find . -name "$1" -exec \  
rm -i {} \; ; }
```

```
bash:~$ type -all rmall
```

```
rmall is a function
```

```
rmall ()
```

```
{  
find . -name "$1" -exec rm -i {} \; ;  
}
```

```
bash:~$ rmall kk
```

```
rm: remove regular file './kk'? y
```

```
bash:~$
```

# Funzioni (5)

- Definire funzioni da file (modificare la shell corrente)

```
bash:~$ more myfunctions
```

```
function rmall () {
```

```
    find . -name "$1" -exec rm -i {} \; ; }
```

```
bash:~$ . ./myfunctions -- o source
```

```
bash:~$ type -all rmall
```

```
rmall is a function
```

```
rmall ()
```

```
{
```

```
find . -name "$1" -exec rm -i {} \; ;
```

```
}
```

```
bash:~$
```

# Builtin `.'` e `source`

- Comandi interni (builtin) della bash
  - equivalenti
  - sintassi
    - `. filename [ arguments ]`
    - `source filename [ arguments ]`
  - entrambi leggono ed eseguono i comandi contenuti in `filename` nell'ambiente della shell corrente

# Funzioni (6)

```
bash:~$ rmall () { find . -name "$1" -exec \
rm -i {} \; ; }
```

- Attenzione a mettere i giusti meccanismi di quoting (escape) per inibire o permettere l'espansione dei metacaratteri da parte della shell !!!!

– " " oppure ' ' oppure \

- Ne parliamo in dettaglio quando ci occuperemo di espansione & quoting

# Funzioni (7)

- **Attenzione:**
  - le variabili definite dentro una funzione sono globali ed accessibili al di fuori della funzione
  - questo spesso crea problemi: vediamo un esempio un po' artificioso



# Funzioni (8)

```
#!/bin/bash
function esempiofun ()
{
    echo in function: $0 $1 $2
    var1="in function"
    echo var1: $var1
}
var1 ="outside function"
echo var1: $var1
echo $0 $1 $2
esempiofun funarg1 funarg2
echo var1: $var1
echo $0 $1 $2
```

# Funzioni (9)

```
bash:~$ ./escript arg1 arg2
var1: outside function
./escript arg1 arg2
in function ./escript funarg1 funarg2
var1: in function
var1: in function
./escript arg1 arg2
bash:~$
```

# Funzioni (10)

- **Attenzione:**
  - le variabili definite dentro una funzioni sono globali ed accessibili al di fuori della funzione
  - questo spesso crea problemi: vediamo un esempio un po' artificioso
  - conviene sempre limitare lo scope delle variabili nelle funzioni con **local**

# Funzioni (11)

```
#!/bin/bash
function esempiofun ()
{ local var1
  echo in function: $0 $1 $2
  var1="in function"
  echo var1: $var1
}
var1 ="outside function"
echo var1: $var1
echo $0 $1 $2
esempiofun funarg1 funarg2
echo var1: $var1
echo $0 $1 $2
```

# Funzioni (12)

```
bash:~$ ./escript arg1 arg2
var1: outside function
./escript arg1 arg2
in function ./escript funarg1 funarg2
var1: in function
var1: outside function
./escript arg1 arg2
bash:~$
```

*Array*

*Minimale...*

# Array in bash

- Sono aggregati di variabili omogenee con un nome
- Sono indicizzati da 0
- Possono essere *sparsi*

# Array in Bash

- Definizioni (alcune possibili):

```
nomi=(pippo pluto paperone)
```

```
nomi[25]=Clarabella
```

- Accesso

```
echo ${nomi[0]}    #stampa pippo
```

```
echo ${nomi[*]}
```

```
echo ${nomi[@]}
```

```
#stampano pippo pluto paperone Clarabella
```



# Array in Bash

- Numero di elementi in un array

```
echo ${#nomi[@]}
```

```
# stampa 4
```

```
# Attenzione!: da non confondere con
```

```
echo ${#nomi}
```

```
# stampa 5 numero dei caratteri del primo  
elemento (pippo)
```

- Cancellazione

```
unset nomi
```

# Esempio: il problema

```
# contenuto di "inputfile"
```

```
pippo
```

```
pluto
```

```
paperone
```

```
minnie
```

```
qui quo qua
```

```
# vogliamo leggerlo, inserirlo in un array e  
scriverlo al contrario in un file  
"ouputfile"
```

# Esempio: lo script

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

```
# apro i file di input ed output
```

```
# descrittori 3 e 4 rispettivamente
```

```
exec 3<inputfile
```

```
exec 4>outputfile
```

# Esempio: lo script

```
#!/bin/bash  
exec 3<inputfile  
exec 4>outputfile  
# lettura file di input (metto ogni linea  
# nell'array l)  
i=0  
while read -u 3 linea; do  
    l[$i]=$linea  
    (( i++ ))  
done
```

```
#!/bin/bash
exec 3<inputfile
exec 4>outputfile
i=0
while read -u 3 linea; do
    l[$i]=$linea
    (( i++ ))
done
#scrittura ....
for ((j=$i-1; j>-1; j--)) ; do
    echo ${l[$j]} 1>&4
```

# Operatori su stringhe

Minimale...

# Sottostringhe

```
${<var>:<offset>}
```

```
${<var>:<offset>:<length>}
```

ritorna la sottostringa di **<var>** che inizia in posizione **<offset>** (NOTA: il primo carattere è in posizione 0)

Nella seconda forma la sottostringa è lunga **<length>** caratteri. Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${A:5}
```

```
illo
```

```
bash:~$ echo ${A:5:2}
```

```
il
```

```
bash:~$
```

# Lunghezza

`${#<var>}`

consente di ottenere la lunghezza (in caratteri) del valore della variabile `<var>` (NOTA: la lunghezza è comunque una stringa)

– Esempio:

```
bash:~$ A=armadillo
```

```
bash:~$ echo ${#A}
```

```
9
```

```
bash:~$ echo ${A:${#A}-4)}
```

```
illo
```

```
bash:~$ B=${A:3:3}
```

```
bash:~$ echo ${#B}          -- $B=adi
```

```
3
```

```
bash:~$
```



# Pattern matching

- È possibile selezionare parti del valore di una variabile sulla base di un pattern (modello)
- I pattern possono contenere \*,?, e [] come per l'espansione di percorso

- Occorrenze iniziali

`${<var>#<pattern>}`

`${<var>##<pattern>}`

se `<pattern>` occorre all'inizio di `$<var>` ritorna la stringa ottenuta eliminando da `$<var>` la più corta / la più lunga occorrenza *iniziale* di `<pattern>`

# Pattern matching (2)

- Occorrenze finali

`${<var>%<pattern>}`

`${<var>%%<pattern>}`

se `<pattern>` occorre alla fine di `${<var>}` ritorna la stringa ottenuta eliminando da `${<var>}` la più corta / la più lunga occorrenza *finale* di `<pattern>`

- esempi:

- `outfile=${infile%.pcx}.gif`

- rimuove l'eventuale estensione `.pcx` dal nome del file (in `infile`) e ci aggiunge `.gif` (`pippo.pcx` → `pippo.gif`)

# Pattern matching (3)

- Esempi (cont):

- `basename=${fullpath##*/}`

- rimuove dal `fullpath` il prefisso più lungo che termina con `'/'` (cioè estrae il nome del file dal path completo)

- `dirname=${fullpath%/*}`

- rimuove dal `fullpath` il suffisso più corto che inizia per `'/'` (cioè estrae il nome della directory dal path completo)

```
bash:~$ fullpath=/home/s/susanna/myfile.c
```

```
bash:~$ echo ${fullpath##*/}
```

```
myfile.c
```

```
bash:~$ echo ${fullpath%/*}
```

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

# Pattern matching (4)

- Esempi (cont):

- **SCRIPTNAME=\${0##\*/}**

- Seleziona dal pathname dello script in esecuzione il nome del file
- Può essere utile per parametrizzare i messaggi stampati es:

```
echo "${SCRIPTNAME}: Error ...."
```

# Sostituzione di sottostringhe

- È possibile sostituire le occorrenze di un pattern nel valore di una variabile

`${<var>/<pattern>/<string>}`

`${<var>//<pattern>/<string>}`

- l'occorrenza più lunga di **pattern** in **var** è sostituita con **string**.
- La prima forma sostituisce solo la prima occorrenza, la seconda le sostituisce tutte
- se **string** è vuota le occorrenze incontrate sono eliminate
- se il primo carattere è **#** o **%** l'occorrenza deve trovarsi all'inizio o alla fine della variabile
- se **var** è **\*** o **@** l'operazione è applicata ad ogni parametro posizionale, e viene ritornata la lista risultante

# Sostituzione di sottostringhe (2)

- Esempi:

```
bash:~$ echo $A
```

```
unEsempioDiSostituzione
```

```
bash:~$ echo ${A/e/eee}
```

```
unEseeeempioDiSostituzione
```

```
bash:~$ echo ${A//e/eee}
```

```
unEseeeempioDiSostituzioneeee
```

```
bash:~$ echo ${A/%e/eee}
```

```
unEsempioDiSostituzioneeee
```

```
bash:~$ ${A/#*n/eee}
```

```
eeeEsempioDiSostituzione
```

```
bash:~$
```

# Scripting c'è molto di più....

- Si può richiedere l'esecuzione di un comando/builtin originale (non ridefinito con funzioni o aliasing con **builtin** e **command**)
- Si possono trattare opzioni sulla riga di comando (builtin **shift**, **getopts**)
- Si può usare il comando **printf** (per la stampa formattata ...)
- è possibile costruire comandi all'interno dello script ed eseguirli (comando **eval**)
- e molto altro ...

# Espansione e Quoting ...



# Espansione e quoting

- *Espansione:*
  - la shell, prima di eseguire la linea di comando interpreta le variabili ed i simboli speciali sostituendoli (espandendoli) con quanto ‘rappresentato’
- *Quoting:*
  - inibizione della espansione per mezzo di simboli che impongono alla shell l’interpretazione ‘letterale’ di altri simboli che altrimenti avrebbero un significato speciale
  - alla fine dell’espansione i simboli di quoting vengono rimossi, in modo che un eventuale programma che riceva il risultato dell’espansione come argomenti non ne trovi traccia

# Vari tipi di espansione

- La *bash*, prima di eseguire un comando opera diverse espansioni, nel seguente ordine:
  1. Espansione degli *alias* e dell'*history*
  2. Espansione delle *parentesi graffe* (C)
  3. Espansione della *tilde* (~) (C)
  4. Espansione delle *variabili* (Korn)
  5. Sostituzione dei *comandi* (Bourne e Korn)
  6. Espansione delle *espressioni aritmetiche*
  7. Suddivisione in *parole*
  8. Espansione di percorso o *globbing*

# Espansione di *alias* ed *history*

- Se la prima parola di una linea di comando è un alias la shell lo espande (ricorsivamente)
  - L'espansione si applica anche alla parola successiva se l'alias termina con spazio o tab
- Se la prima parola inizia con il metacarattere "!" allora la shell interpreta la parola come riferimento alla history es:
  - !n**      *n-esima riga nella history*
  - !!**      *riga di comando precedente*

# Espansione delle *parentesi graffe*

- Meccanismo che permette la generazione di stringhe arbitrarie usando pattern del tipo:
  - `<prefisso>{<elenco>}<uffisso>`
  - l'elenco è dato da una serie di parole separate da virgole
  - ...
  - es:
    - `sal{v,d,modi}are` si espande a `salvare`,  
`saldare`, `salmodiare`
    - `c{{er,as}c,ucin}are` si espande a `cercare`,  
`cascare`, `cucinare`
  - introdotto nella C shell

# Espansione delle *parentesi graffe* (2)

- Ancora es:

```
bash:~$ mkdir m{i,ia}o
```

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

```
miao/ mio/
```

```
bash:~$ rm -f miao/{lib.{?,??},*~,??}.log}
```

```
bash:~$
```

- Nota:

- in questo caso le stringhe che risultano dall'espansione non sono necessariamente nomi di file (come accade invece nell'espansione di percorso)

# Espansione della *tilde* (~)

- Se una parola inizia con il simbolo *tilde* (~)
  - la shell interpreta quanto segue (fino al primo ‘/’), come un username e lo sostituisce con il path della sua home directory
    - `~username` → home directory di `username`
  - ‘~/’ e ‘~’ si espandono nella home directory dell’utente loggato (ovvero nel contenuto della variabile **HOME**)
    - `~/,~` → `$HOME`
  - es.
    - `bash:~$ cd ~besseghi`
    - `bash:/home/personale/besseghi$`

# Espansione delle *variabili*

- In ogni parola del tipo

`$stringa` oppure `${stringa}`

`stringa` viene interpretato come il nome di una variabile e viene espanso dalla shell con il suo valore

es.

```
bash:~$ PARTE=Dani
```

```
bash:~$ echo $PARTEele
```

```
bash:~$ echo ${PARTE}ele
```

```
Daniele
```

```
bash:~$
```

# Sostituzione dei *comandi*

- Consente di espandere un comando con il suo (standard) output:

`$ (<comando>)`

— es.

```
bash:~$ ELENCO=$(ls)
```

```
bash:~$ echo $ELENCO
```

```
pippe pluto paperone main.c
```

```
bash:~$ ELENCO=$(ls *.c)
```

```
bash:~$ echo $ELENCO
```

```
main.c
```

```
bash:~$
```



# Sostituzione dei *comandi* (2)

- Ancora esempi:

*-- rimuove i file che terminano per '~'*

*-- nel sottoalbero con radice in '.'*

```
bash:~$ rm $(find . -name "*~")
```

*-- si può usare una diversa sintassi*

*-- attenzione alla direzione degli apici!!!*

*-- vanno da sin a ds*

```
bash:~$ rm `find . -name "*~" `
```

*-- questa seconda è obsoleta e mantenuta solo per compatibilità ma può spiegare alcuni strani comportamenti*

# Espressioni *aritmetiche*

- Trattamento delle espressioni aritmetiche intere:

`$( <espressione> )` o `$( <espressione> )`

— es.

```
bash:~$ echo 12+23
```

```
12+23
```

```
bash:~$ echo $( (12+23) )
```

```
35
```

*-- dich di variabile intera*

```
bash:~$ let VALORE=$((12+23))
```

```
bash:~$ echo $VALORE + 1
```

```
35 + 1
```

```
bash:~$
```

# Suddivisione in *parole*

- Una parola è una sequenza di caratteri che non sia un operatore o una entità da valutare
  - è una entità atomica (es. arg. fornito ad un programma )
  - I delimitatori di parole sono contenuti nella variabile IFS (*Internal Field Separator*) che per default contiene spazio, tab e newline (‘ ’, ‘\t’, ‘\n’)
  - La suddivisione di parole non avviene per stringhe delimitate da apici singoli e doppi (**quoting**)
  - es.

```
bash:~$ ls "un file con spazi nel nome"  
un file con spazi nel nome  
bash :~$
```

# Suddivisione in *parole* (2)

– es. perché?

```
bash:~$ echo mm${IFS}mm
```

```
mm mm
```

```
bash:~$ echo "mm${IFS}mm"
```

```
mm
```

```
mm
```

```
bash:~$ ls un\ file\ con\ spazi\ nel\ nome
```

```
un file con spazi nel nome
```

```
bash:~$
```

# Espansione di percorso o *globbing*

- Se una parola contiene uno dei simboli speciali ‘\*’, ‘?’ o ‘[...]’
  - viene interpretata come modello ed espansa con l’elenco, ordinato alfabeticamente, dei percorsi (pathname) corrispondenti al modello
  - Nota:
    - l’espansione non riguarda i file nascosti, a meno che il punto ‘.’ non faccia parte del modello:

```
bash:~$ ls .bash*  
.bashrc .bash_profile  
bash:~$
```

# Quoting

- Deriva dal verbo inglese *to quote* (citare) ed indica i meccanismi che inibiscono l'espansione
  - in particolare viene rimosso il significato speciale di alcuni simboli, che nel quoting vengono interpretati *letteralmente*
  - ci sono tre meccanismi di quoting:
    - carattere di escape (backslash) \
    - apici semplici `
    - apici doppi " o virgolette.

# Escape e continuazione

- Il carattere di escape (backslash) \
  - indica che il carattere successivo non deve essere considerato un carattere speciale
  - es:

```
bash:~$ ls .bash\*  
ls: .bash*: No such file or directory  
bash:~$
```

Il modello non é stato espanso e l'asterisco è considerato un carattere normale parte del nome del file da listare
  - *Continuazione*: Se \ è seguito subito dal newline indica che il comando continua sulla linea successiva

# Apici singoli

- Una stringa racchiusa fra apici singoli non è soggetta a *nessuna* espansione

**' testo '**

– attenzione al verso degli apici: l'apice inclinato in modo opposto è legato alla sintassi obsoleta delle sostituzioni dei comandi ( ` )

– es:

```
bash:~$ A=prova
```

```
bash:~$ echo 'nessuna espansione di $A o *'
```

```
nessuna espansione di $A o *
```

```
bash:~$
```



# Apici doppi

- Inibiscono solo l'espansione di percorso:

**"testo"**

– in questo caso \$ e \ vengono valutati normalmente

– es:

```
bash:~$ A=prova
```

```
bash:~$ echo "nessuna espansione di $A \ $A  
o *"
```

```
nessuna espansione di prova $A o *
```

```
bash:~$
```

# Combinare comandi

Una panoramica completa

# Terminazione ed Exit status

- Ogni comando Unix al termine della sua esecuzione restituisce un valore numerico (detto *exit status*)
  - tipicamente *zero* significa esecuzione regolare e ogni altro valore terminazione anomala
  - gli exit status si possono usare nelle espressioni booleane all'interno dei comandi condizionali di shell.
    - in questo caso zero viene assimilato a true e tutto il resto a false.
  - la variabile predefinita `$?` da l'exit status dell'ultimo comando eseguito

# Bash: comandi semplici

`[var assign] <command> <args> <redirs>`

– es: `A=1 B=2 myscript pippo < pluto`

- In pratica:

- è una sequenza (opzionale) di assegnamenti a variabili,
- seguita da una lista di parole di cui la prima (command) è interpretata come il comando da eseguire
- seguita da eventuali ridirezioni (**redirs**)
- terminato da un carattere di controllo (newline o ‘;’)
- L’ *exit status* è quello del comando (se la terminazione è normale) oppure lo stabilisce la shell ...

# Bash: comandi semplici (2)

Codici di terminazione ‘anomala’:

- comando non trovato 127
- file non eseguibile 126
- comando terminato da segnale  $n$ :  $128 + n$
- esempi di evento/segnale /  $n$ 
  - CTRL-C            SIGINT            2
  - **kill**             SIGTERM          15
  - **kill -9**          SIGKILL          9

# Bash: pipelining

```
[ ! ] <command1> [ | <command2> ]
```

- sequenza di comandi separata dal carattere di pipe ‘|’
- In questo caso lo **stdout** di **command1** viene connesso attraverso una pipe allo **stdin** di **command2** etc
- ogni comando è eseguito in un processo differente (sottoshell)
- il suo exit status è quello dell’ultimo comando nella pipeline (o la sua negazione logica se è stato specificato !)

# Liste

- Una lista è una sequenza di una o più pipeline
  - separata da uno degli operatori: `;` `&` `&&` `||`
  - terminata da `;` `&` o *newline*
  - una lista può essere raggruppata da parentesi (tonde o graffe) per controllarne l'esecuzione
  - L'exit status della lista è l'exit status dell'ultimo comando eseguito dalla lista stessa

# Liste: sequenze non condizionali

- Sintassi

**<command1> ; <command2>**

- viene eseguito **command1**

- quando termina **command1** si esegue **command2**

- l'*exit status* è quello di **command2**

- ; sostituisce logicamente il *newline*

```
bash:$ sleep 40; echo done
```

```
-- attende 40 sec
```

```
done
```

```
bash:$
```



# Liste: comando in background

**<command> &**

- la shell esegue **command** in una sottoshell, senza attenderne la terminazione e ripresenta subito il prompt
- l'exit status è 0
- es.

```
bash:$ sleep 40 &
```

```
bash:$
```

# Liste: operatore di controllo & &

- Sintassi:

**<command1> && <command2>**

- la shell esegue **command1**
- se l'exit value di **command1** è 0 (true) esegue anche **command2**
- l'exit value è l'AND logico dell'exit value dei due comandi (lazy)
- serve per eseguire il secondo comando solo se il primo ha avuto successo. Es:

```
bash:$ mkdir prova && echo prova creata!
```

**(segue)**

# Liste: operatore di controllo && (2)

```
bash:$ mkdir prova && echo prova creata!  
prova creata!
```

```
bash:$ mkdir prova && echo prova creata!  
mkdir: cannot create directory `prova': File  
exists  
bash:$
```

# Liste: operatore di controllo | |

- Sintassi:

**<command1> | | <command2>**

- la shell esegue **command1**
- se l'exit value di **command1** è diverso da 0 (false) esegue anche **command2**
- l'exit value è l'OR logico dell'exit value dei due comandi (lazy)
- serve per eseguire il secondo comando solo se il primo *non* ha avuto successo. Es:

```
bash:$ mkdir prova | | echo prova NON creata!  
(segue)
```

# Liste: operatore di controllo || (2)

```
bash:$ mkdir prova && echo prova creata!  
prova creata!
```

```
bash:$ mkdir prova && echo prova creata!  
mkdir: cannot create directory `prova': File  
exists
```

```
bash:$ mkdir prova || echo prova NON creata!  
mkdir: cannot create directory `prova': File  
exists
```

```
prova NON creata!
```

```
bash:$
```

# Delimitatori di lista { ... }

- Sintassi:

```
{ <list>; }
```

- la lista `list` viene eseguita nella shell corrente, senza creare alcuna sottoshell
- L'effetto è quello di raggruppare più comandi in un unico blocco (exit status quello di *list*)
- ATTENZIONE: il `;` finale è necessario come pure lo spazio fra lista e parentesi graffe

```
bash:$ { date; pwd; } > out
```

*-- scrive in 'out' sia l'stdout di date che di pwd*

```
bash:$
```

# Delimitatori di lista (...)

- Sintassi:

( **<list>** )

– la lista **list** viene eseguita in una sottoshell

- assegnamenti di variabili e comandi interni che influenzano l'ambiente di shell non lasciano traccia dopo l'esecuzione
- l'exit status è quello di list

```
bash:$ ( cd Work; mkdir pippo ) && echo OK
```

*-- tenta di spostarsi nella directory Work e di creare la directory pippo, se ci riesce scrive un messaggio di conferma*

```
bash:$
```

# Ridirezione e pipeline

## Approfondimento



# Shell: ridirezione

- Ogni processo Unix ha dei 'canali di comunicazione' predefiniti con il mondo esterno

– es.

```
bash:~$ sort
```

```
pippo
```

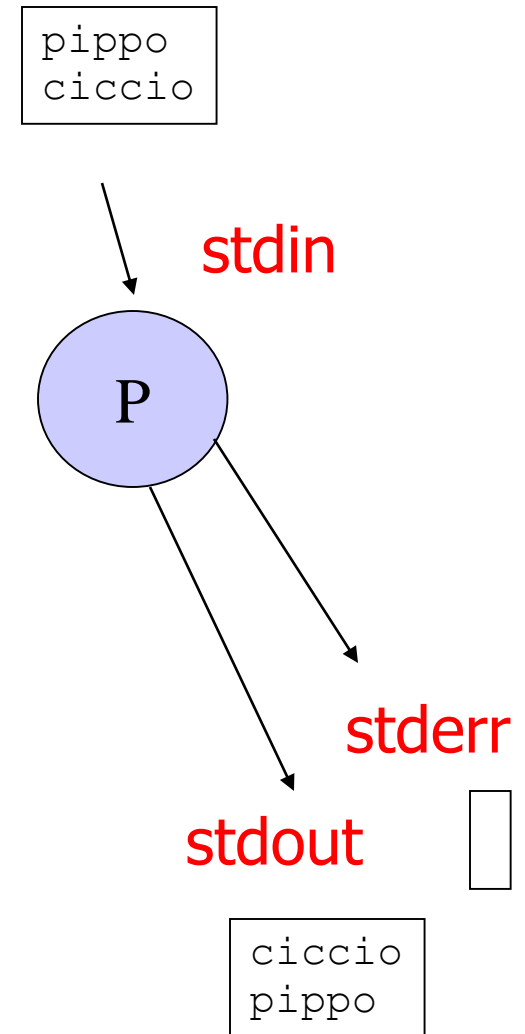
```
ciccio
```

```
^D
```

```
ciccio
```

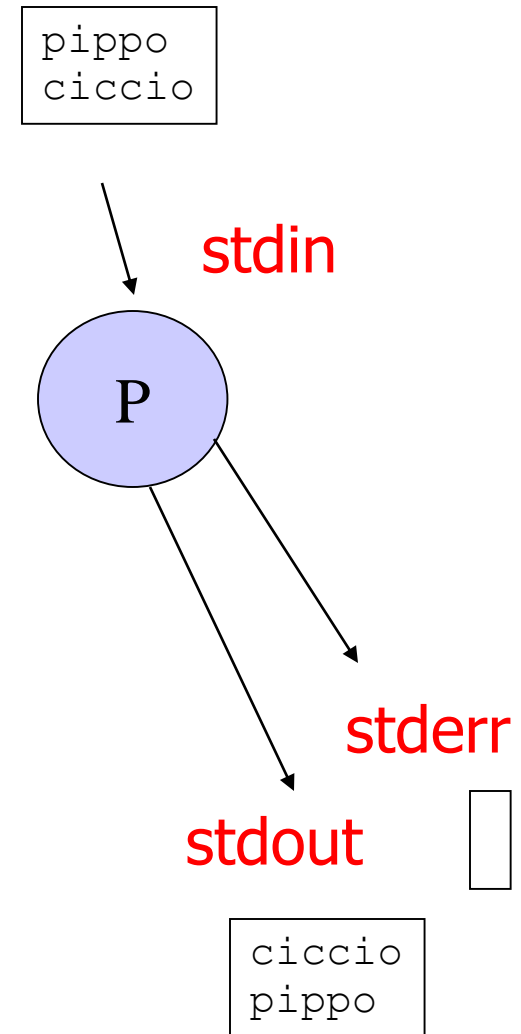
```
pippo
```

```
bash:~$
```



# Shell: ridirezione (2)

- Per default
  - **stdin stdout, stderr** sono associati al terminale di controllo
- La ridirezione (*redirection*) ed il *pipeline* permettono di alterare questo comportamento standard.



# Shell: ridirezione (3)

- Con la ridirezione:
  - **stdin**, **stdout**, **stderr** possono essere collegati a generici file
- Ogni file aperto è identificato da un *descrittore di file* ovvero un intero positivo
- I descrittori standard sono:
  - 0 (**stdin**) 1 (**stdout**) 2 (**stderr**)
  - **n>2** per gli altri file aperti
  - la Bash permette di ridirigere qualsiasi descrittore

# Ridirezione dell'input

- Sintassi generale

**command [n]< filename**

- associa il descrittore **n** al file **filename** aperto in lettura
- se **n** è assente si associa **filename** allo standard input
- Serve anche per aprire un file in lettura (usare un valore di  $n > 2$ ) ad esempio

**exec 3< pippo**

# Ridirezione dell'input (2)

– es.

```
bash:~$sort < lista.utenti
```

```
prog
```

```
root
```

```
susanna
```

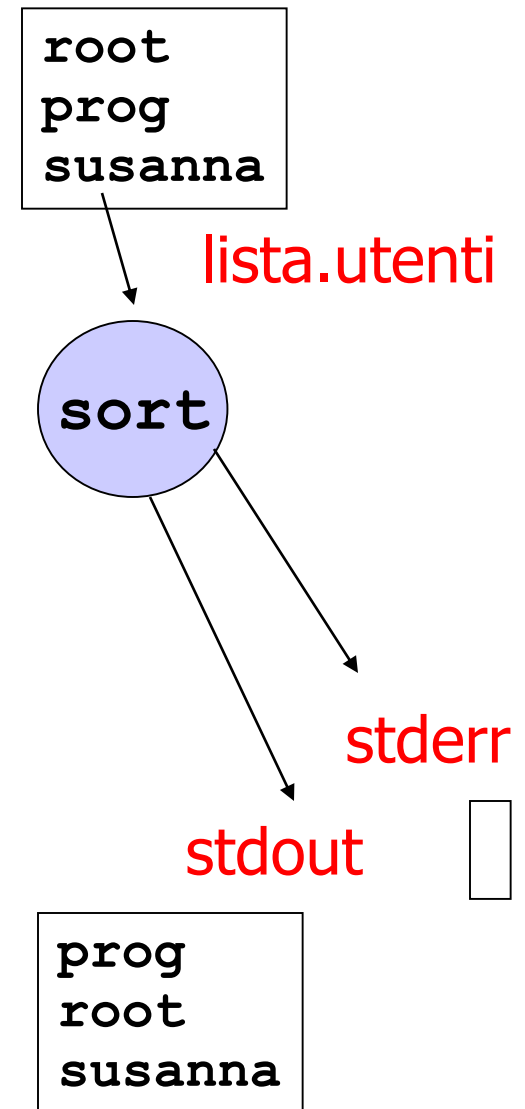
```
bash:~$ sort 0< lista.utenti
```

```
prog
```

```
root
```

```
susanna
```

```
bash:~$
```



# Ridirezione dell'input (3)

– es. lettura file

```
bash:~$ cat leggi.sh
```

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

```
exec 3<lista.utenti
```

```
while read -u 3 linea ; do
```

```
    echo $linea
```

```
done
```

```
bash:~$ ./leggi.sh
```

```
root
```

```
prog
```

```
susanna
```

```
bash:~$
```

```
root  
prog  
susanna
```

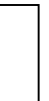
lista.utenti

3

leggi.sh

stdout

```
root  
prog  
susanna
```



# Ridirezione dell'output

- Sintassi generale

**command** [**n**]> **filename**

- associa il descrittore **n** al file **filename** aperto in scrittura
- se **n** è assente si associa **filename** allo standard output

- Attenzione:

- se il file da aprire in scrittura esiste già, viene sovrascritto
- se è attiva la modalità *noclobber* (**set**), ed il file esiste il comando fallisce
- per forzare la sovrascrittura del file, anche se *noclobber* è attivo (**on**) usare '>|'

# Ridirezione dell'output (2)

– esempio

```
bash:~$ ls > dir.txt
```

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

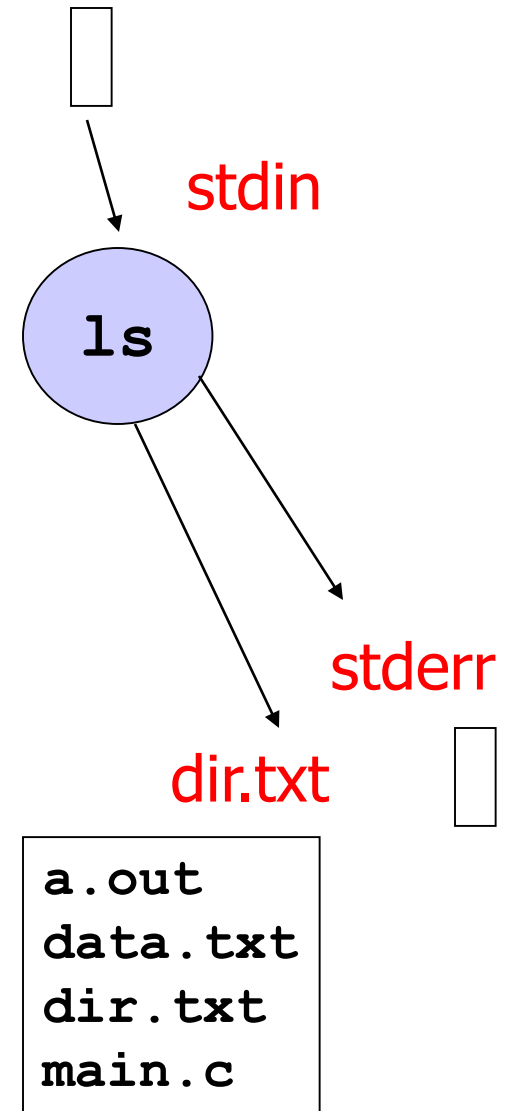
```
a.out
```

```
data.txt
```

```
dir.txt
```

```
main.c
```

```
bash:~$
```





# Ridirezione dell'output (3)

- esempio

```
bash:~$ set -o
```

```
...
```

```
noclobber on
```

```
noexec off
```

```
...
```

```
bash:~$ ls > dir.txt
```

```
-bash: dir.txt: cannot overwrite existing file
```

```
bash:~$ ls >| dir.txt
```

```
bash:~$
```

# Ridirezione dell'output (4)

– es. Lettura/scrittura file

```
bash:~$ cat leggi.sh
```

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

```
exec 3<lista.utenti
```

```
exec 4>pippo
```

```
while read -u 3 linea ; do
```

```
    echo $linea 1>&4
```

```
done
```

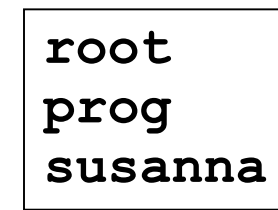
```
bash:~$ ./leggi.sh
```

```
bash:~$ more pippo
```

```
root
```

```
prog
```

```
susanna
```



lista.utenti

3

leggi.sh

4

pippo

root  
prog  
susanna



# Ridirezione dello standard error

- Redirezione dello standard error:

– es.

```
bash:~$ ls dirss.txt
```

```
ls: dirss.txt: No such file or directory
```

```
bash:~$ ls dirss.txt 2> err.log
```

```
bash:~$ more err.log
```

```
ls: dirss.txt: No such file or directory
```

```
bash:~$
```

# Ridirezione dell'output in *append*

- Permette di aggiungere in coda ad un file esistente

**command [n]>>filename**

- associa il descrittore **n** al file **filename** aperto in scrittura, se il file esiste già i dati sono aggiunti in coda
- es.

```
bash:~$ more lista.utenti
```

```
susanna
```

```
prog
```

```
root
```

```
bash:~$ sort < lista.utenti 1>> err.log
```

# Ridirezione dell'output in *append* (2)

– es. (cont)

```
bash:~$ more err.log
```

```
ls: dirss.txt: No such file or directory
```

```
prog
```

```
root
```

```
susanna
```

```
bash:~$
```

# Ridirezione stdout stderr simultanea

`command &> filename` *-- raccomandata*

`command >& filename`

— es.

```
bash:~$ ls CFGVT * &> prova
```

```
bash:~$ more prova
```

```
ls: CFGVT: No such file or directory -- stderr
```

```
a.out -- stdout
```

```
data.txt
```

```
dir.txt
```

```
main.c
```

```
bash:~$
```

# Ridirezione stdout stderr simultanea (2)

– es.

```
bash:~$ ls * CFGVT &> prova
```

```
bash:~$ more prova
```

```
ls: CFGVT: No such file or directory -- stderr
```

```
a.out -- stdout
```

```
data.txt
```

```
dir.txt
```

```
main.c
```

```
bash:~$
```

# Ridirezione: ancora esempi

*-- ridirigo stderr e stdout su due file diversi*

```
bash:~$ ls * CFGVT 1> prova 2>err.log
```

*-- elimino i messaggi di errore*

```
bash:~$ more prova 2> /dev/null
```

*-- ridirigo un descrittore sull'altro*

```
bash:~$ echo Errore!!!! 1>&2
```

```
Errore!!!!
```

```
bash:~$
```



# Ridirezione: *here document*

- Permette di fornire lo standard input di un comando in line in uno script.

– Sintassi: **command << WORD**

**Testo**

**WORD**

- (1) la shell copia in un buffer il **Testo** fino alla linea che inizia con la parola **WORD** (esclusa)
- (2) poi esegue **command** usando questi dati copiati come standard input

# Ridirezione: *here document* (2)

- Esempio:

```
bash:~$ more sulsort.sh
```

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

```
sort << ENDS
```

```
paperone
```

```
minnie
```

```
archimede
```

```
ENDS
```

```
echo Sort finished
```

```
bash:~$
```

# Ridirezione: *here document* (3)

- Esempio (cont):

```
bash:~$ ./sulsort.sh
```

```
archimede
```

```
minnie
```

```
paperone
```

```
Sort finished
```

```
bash:~$
```

# Pipeline

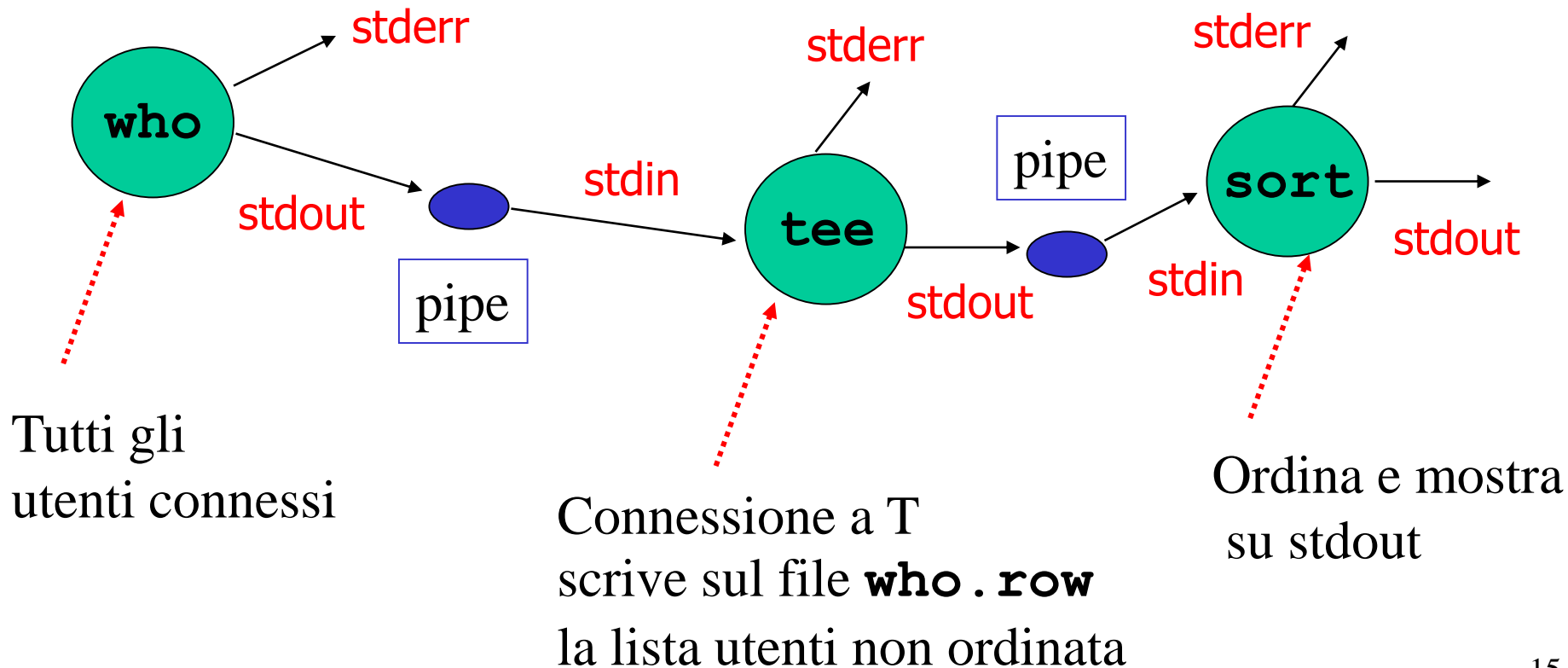
# Bash: pipelining

`<cmd1> | <cmd2> | ... | <cmdN>`

- sequenza di comandi separata dal carattere di pipe ‘|’
- In questo caso lo **stdout** di **command1** viene connesso attraverso una pipe allo **stdin** di **command2** etc
- ogni comando è eseguito in un processo differente (sottoshell)

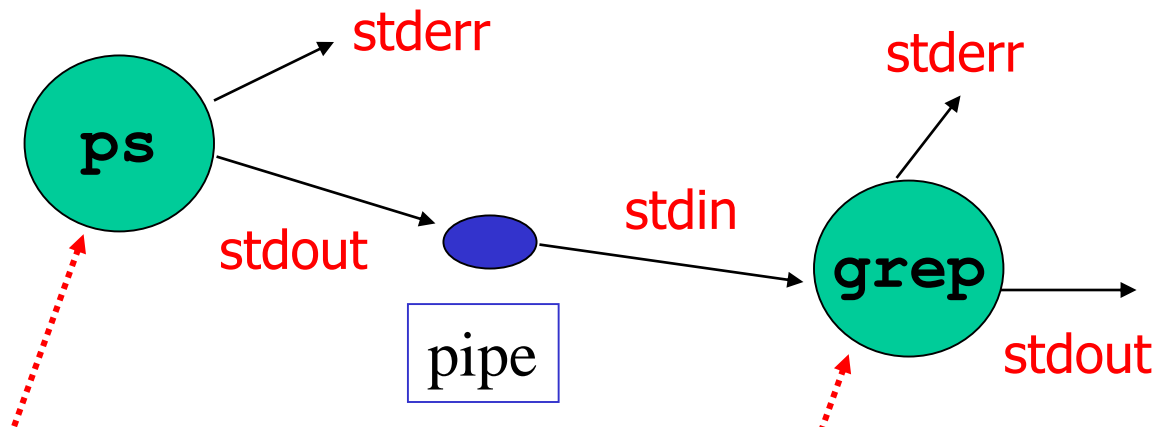
# Pipelining: esempi ...

```
bash:~$ who | tee who.row | sort
```



# Pipelining: esempi ... (2)

```
bash:~$ ps aux | grep ciccio
```



Mostra tutti i processi attivi

Seleziona quelli che contengono  
'ciccio'

Processi ....

Cenni



# Processi

- Cos'è un processo?
  - è un *programma in esecuzione completo del suo stato*
    - dati
    - heap
    - descrittori dei file
    - stack
    - segnali pendenti
    - etc ...

# Processi (2)

- Ci sono comandi che permettono di avere informazioni sui processi attivi
  - centinaia di processi attivi su un sistema Unix/Linux

*-- ps permette di avere informazioni sui  
-- processi attualmente in esecuzione*

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

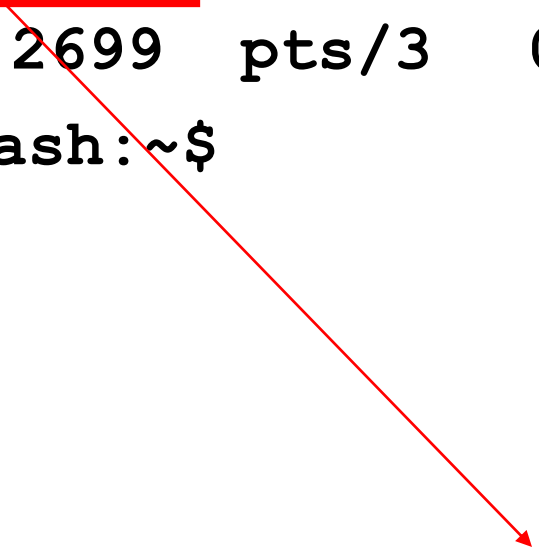
```
bash:~$
```

# Processi (3)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$
```



*PID --Process identifier  
intero che identifica univocamente il processo*

# Processi (4)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$ ls -l /dev/pts/3
```

```
crw--w---- 1 susanna tty 136,3 ..... /dev/pts/3
```

```
bash:~$
```

*Dispositivo  
a caratteri*

*Terminale di controllo*

*Major, minor number  
(Driver, device)*

# Processi (5)

```
bash:~$ ps
```

PID	TTY	TIME	CMD
2692	pts/3	00:00:00	bash
2699	pts/3	00:00:00	ps

```
bash:~$
```

*Tempo di CPU accumulato  
(dd):hh:mm:ss*

*Nome del file  
eseguibile*

# Processi: più informazioni ...

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	1760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

*Status:*

*R -- running or runnable*

*S -- interruptable sleep*

*(wait for event to complete)*

*... molti di più*

# Processi: più informazioni ... (2)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	1760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

*Status:*

*R -- running or runnable*

*S -- interruptable sleep*

*(wait for event to complete)*

*... molti di più*

*System call dove il processo è bloccato*

# Processi: più informazioni ... (3)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	1760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

*Pid del padre*

*Virtual size of process  
text+data +stack*



# Processi: più informazioni ... (4)

```
bash:~$ ps -l
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	1002	2692	1760	0	75	0	-	1079	wait	pts/3	...	bash
0	R	1002	2699	2692	0	76	0	-	619	-	pts/3	...	ps

```
bash:~$
```

*Effective user id*

*%cpu time usato nell'ultimo minuto*

*Scheduling: Priorità, nice*

# Job control ...

Attivare processi in background, etc

# Esecuzione in *background*

- La shell permette di eseguire più di un programma contemporaneamente durante una sessione
- sintassi:

## **command &**

- il comando **command** viene eseguito in background
  - viene eseguito in una sottoshell, di cui la shell non attende la terminazione
  - si passa subito ad eseguire il comando successivo (es. in ambiente interattivo si mostra il prompt)
  - l'exit status è sempre 0
  - *stdin* non viene connesso alla terminale di controllo (un tentativo di input provoca la sospensione del processo)

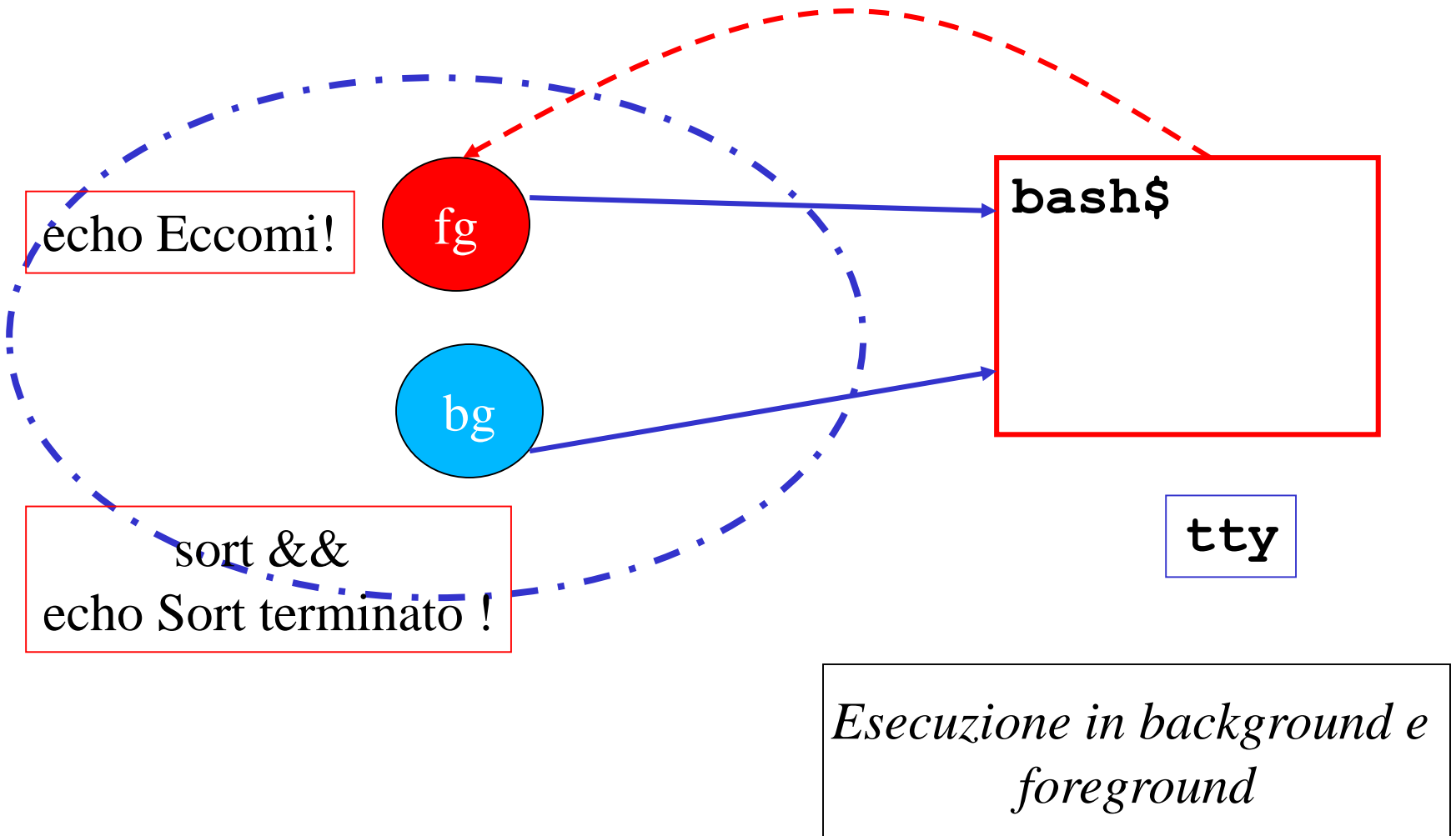
# Esecuzione in *background* (2)

- Esempio:

- processi pesanti con scarsa interazione con l'utente

```
bash:~$ sort <file_enorme >file_enorme.ord \  
&& echo Sort terminato! &  
bash:~$ echo Eccomi!  
Eccomi!  
bash:~$  
Sort terminato!  
bash:~$
```

# Esecuzione in background (3)



*Esecuzione in background e foreground*

# Controllo dei job

- Il builtin **jobs** fornisce la lista dei job nella shell corrente

– un *job* è un insieme di processi correlati che vengono controllati come una singola unità per quanto riguarda l'accesso al terminale di controllo

– es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez2.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

# Controllo dei job (2)

- Il builtin `jobs`...

– es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1] Running      emacs Lez2.tex &
```

```
[2]- Running      emacs Lez3.tex &
```

```
[3]+ Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

*1 numero del job  
diverso dal pid!!! Vedi ps*

*+ job corrente  
(spostato per ultimo da foreground a background)*

# Controllo dei job (3)

- Il builtin `jobs` ...

– es.

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez2.tex &
```

```
[2] -  Running      emacs Lez3.tex &
```

```
[3]+ Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

*- penultimo job corrente  
(penultimo job spostato da foreground a background)*



# Controllo dei job (4)

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez2.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$
```

*Stato:*

*Running -- in esecuzione*

*Stopped -- sospeso in attesa di essere riportato  
in azione*

*Terminated -- ucciso da un segnale*

*Done -- Terminato con exit status 0*

*Exit -- Terminato con exit status diverso da 0*

# Controllo dei job (5)

```
bash:~$ ( sleep 40; echo done ) &
```

```
bash:~$ jobs -l
```

```
[1] 20647 Running      emacs Lez2.tex &
```

```
[2]- 20650 Running      emacs Lez3.tex &
```

```
[3]+ 20662 Running      (sleep 40; echo done) &
```

```
bash:~$
```



*PID della corrispondente sottoshell*

# Terminare i job: **kill**

- Il builtin **kill**

```
kill [-l] [-signal] <lista processi o jobs>
```

- i processi sono indicati con il PID,
- i job da %**numjob** oppure altri modi (vedi man)
- consente di inviare un segnale a un job o un processo
- La gestione di default di quasi tutti i segnali è uccidere il processo che li riceve. Possono però essere personalizzati!

**-- lista dei segnali ammessi**

```
bash:~$ kill -l
```

```
1) SIGHUP  2) SIGINT  ...  9) SIGKILL  .....
```

```
bash:~$
```

# Terminare i job: `kill` (2)

- i processi possono proteggersi da tutti i segnali eccetto SIGKILL (9)

```
bash:~$ jobs
```

```
[1]  Running      emacs Lez2.tex &
```

```
[2]-  Running      emacs Lez3.tex &
```

```
[3]+  Running      ( sleep 40; echo done ) &
```

```
bash:~$ kill -9 %3
```

```
[3]+  Killed ( sleep 40; echo done )
```

```
bash:~$
```

# Sospendere e riattivare un job ...

- CTRL-Z sospende il job in foreground inviando un segnale SIGSTOP

```
bash:~$ sleep 40
```

```
^Z
```

```
bash:~$ jobs
```

```
[1]+  Stopped      sleep 40
```

*-- riattiva il job corrente in background*

*-- inviando un segnale SIGCONT*

```
bash:~$ bg
```

```
bash:~$ jobs
```

```
[1]+  Running      sleep 40
```

```
bash:~$
```

# Sospendere e riattivare un job ... (2)

- CTRL-Z sospende il job in foreground inviando un segnale SIGSTOP

```
bash:~$ sleep 40
```

```
^Z
```

```
bash:~$ jobs
```

```
[1]+  Stopped      sleep 40
```

*-- riattiva il job corrente in foreground*

```
bash:~$ fg
```

```
.....      -- aspetta 40 sec in foreground
```

```
bash:~$
```

# Interrompere un job in foreground

- CTRL-C interrompe il job in foreground inviando un segnale SIGINT

```
bash:~$ sleep 40
```

```
^C
```

```
bash:~$ jobs      -- nessun job attivo
```

```
bash:~$
```

# Gestire i segnali: trap

- Il builtin trap permette di catturare i segnali e personalizzare la loro gestione. Sintassi

```
trap cmd sig1 sig2 ...
```

- significa che all'arrivo di uno qualsiasi fra *sig1 sig2* ... deve essere eseguito *cmd* e poi deve essere ripresa l'esecuzione di ciò che è stato interrotto dall'arrivo del segnale



# Gestire i segnali: trap (2)

– Esempio:

```
bash:~$ less trapscript
```

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

```
trap "echo You hit CTRL-C" INT
```

```
sleep 40
```

```
bash:~$ ./trapscript
```

```
^C
```

```
You hit CTRL-C!
```

```
...
```

```
bash:~$ -- 40 secondo passati
```

```
bash:~$
```

# Gestire i segnali: trap (3)

- Non tutti i segnali possono essere catturati (es: SIGKILL)
- per terminare un processo provare sempre
  - SIGINT (CTRL-C)
  - SIGTERM (inviato di default da **kill** e **killall**)
  - SIGQUIT (CTRL-\)
  - e solo come ultima risorsa SIGKILL (**kill -KILL** oppure **kill -9**)
- per convenzione le applicazioni Unix personalizzano i primi tre per avere una terminazione corretta (rimuovendo file temporanei etc..)
- ci sono anche degli stati in cui i processi sono immuni a SIGKILL ... (vedi Linux scheduler)

# Gestire i segnali: trap (4)

- Per veder tutte le gestioni attive:

```
bash:~$ trap
```

```
trap -- cmd sig
```

```
bash:~$
```

- Per ignorare un segnale si usa il comando vuoto es:

```
trap "" INT
```

- Per tornare alla gestione di default

```
trap - INT
```