

# CORSO BASE SHELL

**Bernardo Damele**

`<bernardo.damele@gmail.com>`

# 1 – Cos'è una shell

- ❑ Shell significa *conchiglia*, racchiude il kernel ed è la superficie con cui l'utente entra in contatto quando vuole interagire con il sistema
- ❑ Fondamentalmente una shell è un *interprete di comandi*, che fornisce all'utente un'interfaccia verso un ricco insieme di utility e un linguaggio di programmazione per “combinare” queste utility
- ❑ Ogni sistema Unix mette a disposizione vari tipi di shell
  - Inizialmente l'amministratore del sistema fornisce all'utente una shell di default che però può essere cambiata (comando `chsh`) e personalizzata

# 2 – Tipi di shell

- ❑ Esiste un certo numero di shell tra cui
  - Bourne shell – La prima shell Unix, fu scritta da *Steve Bourne* all'AT&T Bell Lab
    - Questa shell (`sh`) venne distribuita su molte versioni di Unix
  - Korn shell, `ksh`
  - C shell, `cs`/`tcsh` – La shell di BSD
  - Bourne Again SHell, `bash`
- ❑ Le varie shell presentano numerosi aspetti comuni. Differiscono per la sintassi e per alcune caratteristiche e funzionalità più sostanziali
- ❑ La `bash` è una shell `sh`-compatibile, che ne migliora aspetti interattivi e programmatici. E' la shell di default di tutte le distribuzioni *GNU/Linux*
- ❑ Il file `/etc/shells` contiene la lista delle shell valide

# 3 – Perché usare una shell testuale

- Ormai tutti i sistemi Unix hanno un'interfaccia grafica. Perché usare i comandi in linea di una shell testuale?
  - *Potenza e semplicità* – I comandi Unix sono progettati per risolvere problemi specifici. Sono semplici (senza menù né opzioni nascoste) e proprio per questo potenti (es. `grep pattern filename`)
  - *Velocità e flessibilità* – E' più veloce scrivere pochi caratteri da tastiera piuttosto che cercare un programma opportuno e usare le operazioni che fornisce sulla base delle proprie specifiche esigenze
  - *Accessibilità* – Permette di accedere efficientemente ad un sistema in remoto (es. via `ssh`)

# 4 – Le shell sono programmi eseguibili

- Le shell sono **programmi eseguibili**, possono essere utilizzate in due modi
  - **Interattivo**: Interpreta i comandi immessi da tastiera dall'utente – Interprete dei comandi
  - **Non interattivo**: Interpreta un insieme di comandi scritti dentro un file che prende il nome di *shell script* – Linguaggio di programmazione ad alto livello
  
- Si può avviare qualsiasi shell in ogni momento
  - Esempi
    - \$ sh
    - \$ csh
    - \$ ksh

# 4.1 – Interprete dei comandi

- ❑ La shell è un interprete dei comandi
  
- ❑ Esegue i seguenti compiti
  1. Aspetta che l'utente digiti un comando
  2. Analizza il comando
  3. Trova l'eseguibile per quel determinato comando
  4. Se il comando non viene trovato genera un messaggio di errore
  5. Se invece viene trovato, genera un processo figlio che esegue il comando
  6. Aspetta finchè il comando termini e ne rileva lo stato
  7. Ritorna alla fase 1

# 4.1 – Interprete dei comandi

- Come scrivere un comando lungo senza ricorrere ad uno shell script
  - Per continuare un comando su una nuova linea si usa il carattere backslash (\) alla fine della linea di comando
    - Esempio

```
$ cd \  
> /tmp
```
  
- Comandi raggruppati
  - Per raggruppare una serie di comandi si usano le parentesi tonde ( ), ogni gruppo di comandi viene chiamato *espressione*
  - La shell crea una *sotto-shell* per ogni espressione
    - Esempio

```
( cd / ; ls ) || ( pwd ; id )
```

## 4.2 – Linguaggio di programmazione

- ❑ La shell è un linguaggio di programmazione, con variabili, funzioni e costrutti per il controllo del flusso
  
- ❑ Perché utilizzare uno shell script?
  - E' un modo facile e veloce per eseguire una serie complessa di operazioni oppure la stessa operazione più volte, ciclicamente
  
- ❑ Creare uno **shell script**
  - Uno shell script è un file che contiene uno o più comandi che la shell esegue
    - I comandi possono essere di tre tipi
      - Comandi interni – eseguibili di default (es. `ls`, `cd`, `pwd`, `grep`...)
      - Comandi esterni – eseguibili esterni (es. `awk`, `less`, `display`...)
      - Altri shell script
    - Comandi detti *costrutti* per il controllo del flusso

## 4.2 – Linguaggio di programmazione

- ❑ Per far riconoscere al sistema che un file è uno shell script bisogna anteporre a tutti i comandi `#!/bin/bash`. Questo è l'*header* che identifica univocamente che il file è uno script bash
  
- ❑ Per inserire una linea di commento all'interno di uno shell script si usa il carattere cancelletto (`#`). Questa linea non sarà interpretata dalla shell in fase di esecuzione. Serve solo a facilitare la lettura del sorgente stesso
  
- ❑ Rendere un file eseguibile
  - Quando si crea uno shell script usando un editor di testo (es. nano), ha permessi di esecuzione?  

```
[user@hostname userhomedir ]$ ./nomescript  
bash: ./nomescript: Permission denied.
```

```
[user@hostname userhomedir ]$ chmod +x nomescript
```

## 4.2 – Linguaggio di programmazione

- Per eseguire uno shell script, dopo averlo scritto, salvato e reso eseguibile, lo si richiama dalla linea di comando

```
[user@hostname userhomedir ]$ ./nomescript  
questa è una prova
```

- La shell interpreta linea per linea ed esegue un nuovo processo
  - Questo processo è una doppia copia del processo della shell, si chiama sotto-shell
- Il nuovo processo esegue i comandi
  - Se il comando è un eseguibile e la sintassi è corretta
    - L'esecuzione va a buon fine – *exit status 0*
    - Il sistema si occupa di reallocare la sotto-shell al comando successivo

# 5 – Comandi della shell

- La sintassi tipica dei comandi è la seguente

```
$ comando [opzioni] [argomenti]
```

- Opzioni

- Sono opzionali e influiscono sul funzionamento del comando
- Consistono generalmente di un trattino (-) seguito da una sola lettera (es. `ls -l`). Hanno anche una forma estesa (es. `ls --format long`). Possono essere seguite da un argomento (es. `ls -l /tmp`). Spesso più opzioni possono essere raggruppate insieme dopo un solo trattino (es. `ls -lah`)

- Argomenti

- Si possono avere nessuno o più argomenti, dipende dal comando
- Alcuni argomenti sono opzionali. Se non specificati assumono valori di default

# 5.1 – Esempi di comandi della shell

- *Nessun argomento* – Il comando `date` mostra data e ora corrente
  - `date`
  
- *Un solo argomento* – Il comando `cd` cambia la directory di lavoro corrente in quella specificata nel suo argomento
  - `cd /tmp`
  
- *Un'opzione ed un argomento* – Il comando `wc` conta il numero di parole/caratteri/linee in un file, in base all'opzione specificata
  - `wc -w nomefile`                      conta le parole in `nomefile`
  - `wc -m nomefile`                      conta i caratteri in `nomefile`
  - `wc -l nomefile`                      conta le linee in `nomefile`

# 5.1 – Esempi di comandi della shell

- ❑ *Numero arbitrario di argomenti* – Il comando `cat` mostra a video il contenuto dei file di testo passati come argomenti

- `cat nomefile1 nomefile2 nomefile3`

- ❑ *Più opzioni e un argomento di default*

- `ls -lah`

Lista dettagliata (`l`, long) di tutti (`a`, all) i file, anche quelli nascosti con (`h`, human readable) dimensione espressa in Kb, Mb, Gb o Tb. L'argomento di default è la directory di lavoro corrente

## 5.2 – Alcuni comandi della shell

- Alcuni dei comandi interni della bash
  - `cd` – cambia la directory corrente
  - `cp` – copia un file
  - `echo` – mostra a video ciò che gli viene passato come argomento
  - `exit` – esce dalla shell
  - `ls` – mostra i file contenuti nella directory corrente
  - `mkdir` – crea una directory
  - `mv` – sposta (o rinomina) un file
  - `ps` – mostra i comandi in esecuzione
  - `pwd` – mostra a video la directory corrente
  - `rm` – elimina un file
  - `rmdir` – elimina una directory
  - `touch` – crea un file di testo o cambia la data di un file

## 5.2 – Alcuni comandi della shell

### □ Altri comandi della bash

- `awk` – ricerca ed elabora pattern in un file
- `cal` – mostra a video il calendario
- `cat` – mostra a video i file di testo passati come argomenti
- `cmp` – confronta il contenuto di due file
- `cut` – estrae colonne da un file di testo
- `date` – mostra a video e imposta la data/ora del sistema
- `diff` – mostra a video le differenze tra due file di testo
- `df` – mostra la quantità di spazio disponibile sul disco
- `du` – mostra informazioni sullo spazio utilizzato sul disco
- `file` – identifica un file
- `find` – ricerca un file sul disco

## 5.2 – Alcuni comandi della shell

- Qualche altro comando della bash
  - `fsck` – controlla ed eventualmente ripara il file system
  - `grep` – ricerca pattern in un file di testo
  - `gzip` – comprime un file
  - `head` – mostra le righe iniziali di un file di testo
  - `ln` – crea un collegamento ad un file
  - `lp(r)` – stampa un file
  - `lprm` – rimuove un processo dalla coda di stampa
  - `more` – mostra il contenuto di un file di testo una schermata per volta
  - `sed` – editor di stream
  - `sort` – ordina e unisce file
  - `tail` – mostra a video le ultime righe di un file

## 5.2 – Alcuni comandi della shell

- Ancora qualche comando della bash
  - `uniq` – rimuove duplicati in un file di testo
  - `w` – mostra a video gli utenti collegati alla macchina
  - `wc` – conta le righe, le parole ed i caratteri di un file
  - `whatis` – elenca le pagine di manuale per un comando
  - `whereis` – mostra il percorso completo di un eseguibile
  - `which` – localizza un eseguibile usando la variabile di ambiente `PATH`

# 6 – Caratteri di controllo

- Alcune combinazioni di tasti hanno un effetto particolare sul terminale. I **caratteri di controllo** sono ottenuti premendo il tasto `CTRL` (indicato per convenzione con il carattere `^`) e un secondo tasto contemporaneamente. Ecco i più comuni
  - `^C (CTRL-C)` – Interrompe il processo in esecuzione in foreground
  - `^D (CTRL-D)` – Esce dal programma interattivo, corrisponde al segnale *EOD*
  - `^H (CTRL-H)` – Solitamente è equivalente al *backspace*
  - `^Q (CTRL-Q)` – Sblocca lo scorrimento della shell
  - `^S (CTRL-S)` – Blocca lo scorrimento della shell
  - `^U (CTRL-U)` – Cancella la linea di comando
  - `^Z (CTRL-Z)` – Sospende il processo in esecuzione in foreground

# 6.1 – Esempio caratteri di controllo

- Il comando `cat`, senza argomenti, copia l'input (battuto da tastiera) sull'output (video)
  - Esempio

```
$ cat
questa e' una prova          <- input
questa e' una prova          <- output
```
  - La mancanza del prompt indica che siamo ancora in `cat`
- Per segnalare a `cat`, e a molti altri comandi interattivi, la fine dell'inserimento si fa ricorso alla combinazione `^D`, detto, in questo caso, *carattere di fine file* (**EOF**)

# 7 – Editing della linea di comando

- Molte shell, tra cui la `bash`, offrono funzioni di **editing della linea di comando**, ottenute anch'esse premendo il tasto `CTRL` e un secondo tasto contemporaneamente. Ecco alcune delle funzionalità più utili
  - `^A (CTRL-A)` – Va a inizio linea
  - `^E (CTRL-E)` – Va a fine linea
  - `^K (CTRL-K)` – Cancella il resto della linea
  - `^Y (CTRL-Y)` – Reinscrive la stringa cancellata
  - `^D (CTRL-D)` – Cancella il carattere sul cursore
  
- La shell registra i comandi inseriti dall'utente. Il comando `history` li elenca. Per richiamarli
  - `!!` – Richiama il comando precedente
  - `!n` – Richiama l'n-esimo comando

# 8 – Completamento dei comandi

- Un'altra caratteristica della `bash` è il **completamento automatico della linea di comando** premendo il tasto `TAB`

- Esempio

Supponiamo di essere in questa situazione

```
$ ls  
un_file_con_nome_molto_lungo
```

Per copiare il file su `corto`, anziché digitare per esteso il nome si può digitare `cp un,` quindi premerendo il tasto `TAB` il resto del nome viene mostrato nella linea di comando e si potrà a questo punto completare il comando

- Se esistono più completamenti possibili premendo consecutivamente due volte il tasto `TAB` verrà visualizzata a video la lista di tutti i file che iniziano con i caratteri digitati

# 9 – Caratteri speciali

- Quando si digita un comando la shell ricerca i caratteri speciali, detti anche *metacaratteri*
  - Una volta trovati la shell provvede a rimpiazzarli
    - Esempio
      - \$ echo \*
      - Non mostrerà a video il carattere \*
  - Per usare un carattere speciale come testo semplice bisogna scriverlo
    - Usando gli apici singoli: echo '\*'
    - Usando i doppi apici: echo "\*"
    - Usando un unico backslash: echo \\*

# 9 – Caratteri speciali

❑ **# echo a \" is a special character**  
a " is a special character

```
# echo a " is a special character
>
>here it is waiting for a closing "
a is a special character
```

here it is waiting for a closing

**# echo a ''' is a special character**  
a " is a special character

**# echo showing multiple \\ \\ \\ \\ is tricky**  
showing multiple \\ \\ \\ is tricky

**# echo then again maybe not '\\ \\ \\'**  
then again maybe not \\ \\ \\

**# echo then again maybe not "\\ \\ \\"**  
then again maybe not \\

# 9.1 – Lo spazio è un carattere speciale

- Lo spazio è un carattere speciale
  - Cosa implica scriverlo tra apici?

· Esempi

```
$ echo Hello there
Hello there
```

```
$ echo Hello     there
Hello there
```

```
$ echo "Hello     there"
Hello     there
```

# 10 – Operatori di controllo

- Gli operatori di controllo sono una serie di caratteri ( ; & && | | ) che alterano il risultato di un comando
  - Eseguire un comando in background
    - Esempio

```
$ ls &
```
  - Il carattere ; può essere usato per digitare (quindi eseguire dalla shell) più di un comando su una singola linea
    - Esempio

```
$ cd / ; echo sono alla radice ; ls
```

# 10 – Operatori di controllo

- L'operatore `&&` (*and*) dice alla shell di eseguire il secondo comando solo se il primo è terminato e ha restituito al processo padre un *exit status 0*, ovvero senza errori

- Esempio

```
$ grep root /etc/passwd && echo "trovato"
```

```
$ grep qlc /etc/passwd && echo "non trovato"
```

- L'operatore `||` (*or*) dice alla shell di eseguire il secondo comando solo se il primo è terminato e ha restituito al processo padre un *exit status diverso da 0*

- Esempio

```
$ grep root /etc/passwd || echo "non trovato"
```

```
$ grep qlc /etc/passwd || echo "non trovato"
```

# 11 – Redirezione dell'I/O

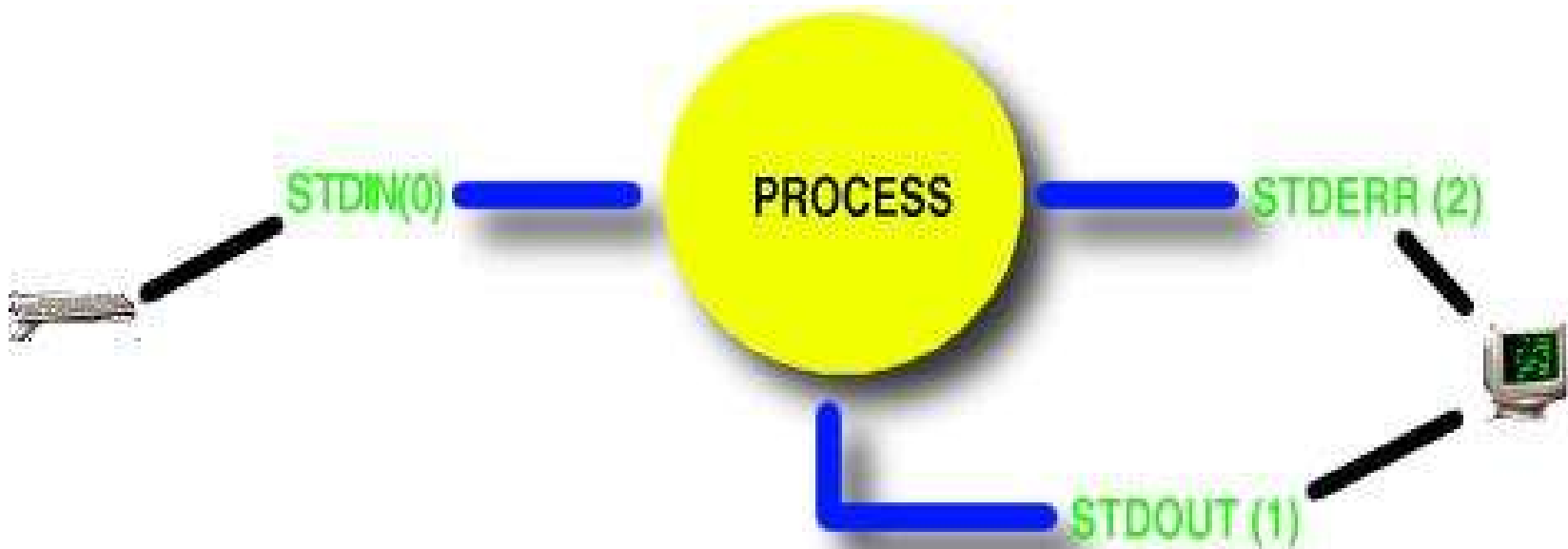
- ❑ E' un modo per mandare input e output (I/O) da diverse parti
  
- ❑ Le redirezioni sono usate per
  - Leggere del testo da un file – Redirezione dello standard input
    - Esempio: `cat < /etc/passwd`
  
  - Scrivere del testo su un file – Redirezione dello standard output (o error)
    - Esempio: `ls -lR > tutti.miei.files`
  
  - Unire più comandi assieme – Redirezione dello standard output sullo standard input del comando successivo
    - Esempio: `grep "/bin/bash" /etc/passwd | cut -f1 -d":"`

# 11.1 – Come funziona la redirectione I/O

- ❑ Ogni processo ha una tabella di *file descriptors*, uno per ogni file
- ❑ Ogni processo ha tre *file descriptors* che sono usati di default

Nome	File descriptor	Destinazione di default
Standard input (stdin)	0	Tastiera
Standard output (stdout)	1	Monitor
Standard error (stderr)	2	Monitor

# 11.1 – Come funziona la redirectione I/O



# 11.2 – Redirezione I/O e caratteri speciali

- La redirezione dell'I/O usa alcuni caratteri speciali

Redirezione I/O	Risultato
comando < file	Prende lo standard input da file
comando > file	Scrive l'output di comando in file. Sovrascrive ogni cosa già presente sul file
comando >> file	Appende l'output di comando su file
comando << WORD > testo > WORD	Prende lo standard input per comando dalle linee che seguono a video fino a WORD. E' chiamato <b>here document</b>
comando1   comando2	Passa l'output di comando1 all'input di comando2. E' chiamato <b>pipeline</b>
comando >& file_descriptor	Redirige l'output di comando ad un <i>file descriptor</i> (file_descriptor, espresso in numero)

## 11.3 – Esempi sulla redirectione dell'I/O

- ❑ Redirigere lo standard output del comando `ls -lR /home/<user>` sul file `tutti.files` e lo standard error su `/tmp/errors`

```
$ ls -lR /home/<user> 1> tutti.files 2> /tmp/errors
```

- ❑ Redirigere lo standard output e lo standard error del comando `ls -lR /tmp` sul file `output.e.error`

```
$ ls -lR /tmp 1> output.e.error 2>&1
```

- ❑ Redirigere lo standard output del comando `ls -l /tmp` sullo standard input del comando `grep root`

```
$ ls -l /tmp | grep root
```

# 11.4 – Redirezione dell'I/O e i device file

- UNIX tratta tutto (devices, processi, kernel) come file
  - Esempi
    - Il comando `tty` mostra il device file associato al terminale (shell) in uso

```
$ tty
```
    - Mandare l'output del comando `ls` ad un terminale (shell di un utente)

```
$ ls > /dev/pts/#
```
    - Mandare il contenuto di un file nel *nulla*

```
$ cat file > /dev/null
```

# 11.5 – Esercizi sulla redirectione I/O

- Redirigere l'I/O su un file
  - Creare un file chiamato `tutti.miei.files` che contiene l'output del comando `ls`
  - Aggiungere il messaggio "FINE DEL FILE" alla fine del file `tutti.miei.files`
  
- Pipelines
  - Contare il numero di files nella propria directory

# 12 – Variabili della shell

## □ La shell supporta le variabili

- Una variabile è un nome (“contenitore”) al quale è associato un *valore*
- Modi per usare le variabili

### · Assegnamento di un valore

- Sintassi: `<nome_variabile>=[valore]`
- Esempio: `nome="linus torvalds"`

Se la variabile `nome` non esiste allora viene creata, altrimenti il suo valore precedente viene sovrascritto

### · Accedere al valore

- Quando la shell vede il carattere dollaro (\$) seguito dal nome di una variabile la rimpiazza col suo valore
- Esempio  
`echo $nome`

# 12 – Variabili della shell

## □ I nomi delle variabili

- Devono iniziare con una lettera o con il carattere *underscore* (`_`)
- Carattere seguito da 0 o più lettere, numeri o *underscores* (`_`)
- Le parentesi graffe (`{ }`) in `bash` servono per delimitare il nome di una variabile da altri caratteri che non fanno parte del nome della variabile

### · Esempi

```
$ nome=
```

```
$ nome2=" "
```

```
$ echo linus${nome}torvalds
```

```
linustorvalds
```

```
$ echo linus${nome2}torvalds
```

```
linus torvalds
```

```
$ echo linus${nome3}torvalds
```

```
linustorvalds
```

# 12 – Variabili della shell

- Una variabile si dice definita quando contiene un valore. Può essere cancellata con

```
$ unset nome_variabile
```

- Esempio

\$ Y = terra	assegna il valore 'terra' alla variabile Y
\$ echo \$Y	stampa sullo standard output il valore della variabile Y (terra)
\$ X = \$Y	assegna il valore della variabile Y (terra) alla variabile X
\$ X = Y	assegna il valore 'Y' alla variabile X
\$ Z =	assegna la stringa vuota alla variabile Z
\$ unset Y	cancella la variabile Y

# 12 – Variabili della shell

- E' possibile assegnare ad una variabile il risultato di un comando in due modi (vedasi la slide intitolata “*Sostituzione del comando*”)

- Facendo uso dell'apice inverso (^)

- Esempio

```
$ VAR=`ls`  
$ echo $VAR
```

- Facendo uso del dollaro (\$) e il comando tra *parentesi tonde*

- Esempio

```
$ VAR=$(ls)  
$ echo $VAR
```

# 12.1 – Variabili locali e variabili di ambiente

- Di default le variabili sono **locali**, in quanto valide soltanto all'interno del processo dove sono state inizializzate. Queste non sono quindi passate ad altri processi figli
  
- Per far sì che una variabile locale sia una **variabile di ambiente**, quindi *globale* e richiamabile da altri processi figli si usa il comando `export`
  - Esempio

```
$ CLASSPATH=$NETSCAPE_HOME/classes
$ export CLASSPATH
```
  
- In questo modo nuovi processi e sotto-shell ereditano tali variabili dalla shell che li ha inizializzate ed esportate

# 12.1 – Variabili locali e variabili di ambiente

- Alcune variabili sono inizializzate automaticamente dalla shell. Tipicamente hanno il nome costituito da lettere maiuscole
  - SHELL contiene il pathname completo della shell di default
  - HOME contiene il pathname completo della propria home directory
  - PWD contiene il valore della working directory
  - ...
  
- Quali variabili sono impostate
  - Il comando `set`, senza alcun argomento, visualizza la lista delle variabili locali assegnate
  - Il comando `env` visualizza la lista delle variabili di ambiente

# 13 – Parametri di posizione

- Il nome del comando ed i suoi argomenti sono chiamati **parametri di posizione**
  - Ci si fa riferimento tramite la loro posizione nella linea di comando
  - \$0: Nome del programma
  - \$1 – \$9: Argomenti del programma
    - Il primo argomento è rappresentato da \$1
    - Il secondo argomento è rappresentato da \$2
    - E così via fino a \$9
    - Nel caso un comando abbia più di nove argomenti essi devono essere shiftati per poter riutilizzare le posizioni da \$1 a \$9
  - Esempio

```
$ ls -l /tmp
```

ls è il primo parametro (\$0), -l è il secondo (\$1) e /tmp è il terzo (\$2)

# 13.1 – Parametri speciali

- Exit status: \$ ?
  - Quando un processo termina ritorna un **exit status** al suo processo padre
  - Per convenzione
    - *Il valore 0* rappresenta la buona riuscita del comando
    - *Ogni altro valore* rappresenta il fallimento del comando
  - Si può specificare l'exit status che uno shell script ritorna alla fine della sua esecuzione tramite il comando `exit` seguito da un numero
    - Altrimento l'exit status dello script corrisponde all'exit status dell'ultimo comando che lo script esegue

## 13.2 – Altri parametri speciali

- Il valore degli argomenti sulla linea di comando: \$\* e \$@
  - \$\* e \$@ rappresentano tutti gli argomenti sulla linea di comando (non solo i primi nove)
  - \$\*: Tratta la lista di argomenti come un singolo argomento
  - \$@: Produce una lista di argomenti separati
- Il numero di argomenti sulla linea di comando: \$#
  - Ritorna un numero decimale
- Il pid della shell: \$\$
  - Ritorna un numero decimale
- Il pid dell'ultimo processo che è stato avviato in background: \$!
  - Ritorna un numero decimale

# 14 – Sostituzione del comando

- La sostituzione del comando permette di inserire l'output di un comando nella linea di comando di un altro comando, di una variabile o di un costrutto
  - La sostituzione del comando si esegue con l'apice inverso (`), oppure con il dollaro (\$) e il comando tra parentesi tonde
  - Esempi

```
$ echo `ls`
```

oppure

```
$ echo $(ls)
```

# 15 – Espansione del pathname

- Un modo per specificare un numero di files con un piccolo numero di caratteri
  - Anche conosciuto come *filename substitution* o *globbing*
  - Esempi
    - Il metodo standard per visualizzare tutti i file word all'interno di una dir è  

```
$ ls -l word.doc fred.doc chap1.doc chap2.doc
```
    - Il metodo che si basa sull'espansione è  

```
$ ls -l *.doc
```
  - Anche le espansioni della shell si appoggiano a certi caratteri speciali
    - \*
      - Con l'asterisco si espande ogni stringa
    - ?
      - Con il punto di domanda si espande un singolo carattere
    - [caratteri]
      - Espande una serie di caratteri tra parentesi quadre
      - Due caratteri separati da un trattino (-) indicano un range di caratteri
      - Se ! o ^ sono il primo carattere della sequenza allora vengono espansi tutti quelli diversi dai caratteri successivi o da quel range di caratteri

# 15.1 – Altre espansioni

- La bash supporta altre espansioni che possono rivelarsi utili

- Espansione delle *parentesi graffe*

- Simile all'espansione pathname, ma non espande dei veri files

- Esempio

- ```
$ ls -l /etc/rc.d/{init,rc1,rc2}*
```

- Espansione della *tilde*

- Il carattere tilde (~) espande la home directory, la directory corrente o la directory utilizzata precedentemente in base al carattere successivo

- Esempi

- ```
$ echo la mia home directory: ~
```

- ```
$ echo la mia directory attuale: ~+
```

- ```
$ echo la precedente directory era: ~-
```

- Espansioni *aritmetiche*

- Esempio

- ```
$ echo $[(5+4)-3*2]
```

# 16 – Struttura condizionale

- In bash la *struttura condizionale* si realizza con il costrutto `if`. La sintassi è la seguente

```
        if test espressione1
then
            comandi1

[      elif test espressione2
then
            comandi2      ]

[      else
            comandi3      ]
fi
```

- La bash esegue l'espressione (*espressione1*), se il suo *exit status* è 0 (nel caso l'espressione risulti essere vera) esegue il blocco di `comandi1` altrimenti esegue la seconda espressione e così via

# 16.1 – Comando test

- Il comando `test` è parte integrante del costrutto `if`. Ci sono due modi per utilizzarlo

- Modo esplicito

- Esempio

```
if test $(ls $HOME | wc -l) -gt 2
then
    echo "ci sono piu' di due file in $HOME"
fi
```

- Modo implicito

- Esempio

```
if [ $(ls $HOME | wc -l) -gt 2 ]
then
    echo "ci sono piu' di due file in $HOME"
fi
```

# 17 – Strutture cicliche

- In bash le *strutture cicliche* si utilizzano per realizzare un ciclo. Sono quindi utili per eseguire una simile operazione più volte ciclicamente. Le strutture cicliche si possono realizzare con tre diversi costrutti

- Il costrutto `for` con la sintassi che segue

```
for VAR in espressione  
do  
    comandi  
done
```

- Esempio

```
for X in $(ls)  
do  
    echo "$X"  
done
```

# 17 – Strutture cicliche

- Il costrutto `while` con la sintassi che segue

```
while test espressione
do
    comandi
done
```

- Esempio

```
X=1
while test $X -le 10
do
    echo "Sono a $X"
    X=$((expr $X + 1))
done
```

# 17 – Strutture cicliche

- Il costrutto `until` con la sintassi che segue

```
until test espressione  
do  
    comandi  
done
```

- Esempio

```
X=1  
until test $X -gt 10  
do  
    echo "Sono a $X"  
    X=$((expr $X + 1))  
done
```