

bash

Lorenzo Micheli

<lorenzo.micheli@gmail.com>

28 giugno 2011

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

# La Shell

- ▶ La **Shell** un programma che fornisce un tipo di interfaccia per l'utente, che permette di accedere ai servizi offerti dal kernel del sistema operativo.
- ▶ Le Shell testuali sono degli interpreti di comandi, che oltre a fornire un'interfaccia tra l'utente ed il sistema operativo ricca di utility, offrono le potenzialità di veri e propri linguaggi di programmazione.

## Prospettiva storica

Unix fu' sviluppato partendo dall'idea che l'utente avrebbe dovuto essere in grado di interagire direttamente con il computer. Per fornire questa capacita' all'utente, gli sviluppatori progettarono un programma che leggesse un comando da input e lo eseguisse, la Shell.

- ▶ La **Shell Thompson** fu' la prima Shell di Unix, introdotta nella prima versione di Unix nel 1971, scritta da Ken Thompson. La Shell Thompson era un semplice interprete di comando con poche funzionalita', senza alcun supporto per lo scripting.
- ▶ La **Shell Bourne**, o **sh**, divenne la shell di default di Unix a partire dalla versione 7, rimpiazzando la Shell Thompson. Sviluppata da *Stephen Bourne* presso i *Bell Laboratories* della *AT&T*, e fu' rilasciata nel 1977.

## Prospettiva storica

- ▶ La **C Shell**, o **cs***h*, sviluppata da **Bill Joy** per il sistema Unix **BSD**. La sintassi della C Shell era simile a quella del linguaggio di programmazione C.
- ▶ La **Shell Korn**, o **ksh**, scritta da *David Korn* presso i *Bell Labs*, nei primi anni 80. Compatibile con la Shell Bourne, la ksh include molte features della C Shell.
- ▶ La **Bourne-Again Shell**, o **bash** e' fondamentalmente una shell che associa le caratteristiche della Shell C e della Shell Korn. Bash e' stata sviluppata dalla *Free Software Foundation* nel 1987 e fa parte del progetto **GNU**. Bash e' la shell di default di tutti i sistemi GNU/Linux.

# Invocare la shell

Le Shell sono **programmi eseguibili**, e come tali possono essere invocate da linea di comando, esempio:

```
% sh
% csh
% ksh
% bash
```

Le Shell possono essere invocate in due modi:

- ▶ **Interattivo**: interpreta i comandi immessi da linea di comando dall'utente
- ▶ **Non-Interattivo**: interpreta uno o più comandi contenuti all'interno di un file, tale file prende il nome di **Shell Script**

## Modalita' Interattiva

La Shell, invocata in modalita' interattiva, e' un interprete che legge i comandi forniti dall'utente attraverso la tastiera.

La shell in modalita' interattiva esegue i seguenti compiti:

1. Aspetta un'input da parte dell'utente
2. Analizza il comando
3. Trova l'eseguibile del comando
4. Se il comando non viene trovato genera un messaggio di errore
5. Se il comando viene trovato genera un processo figlio che esegue il comando
6. Terminato il comando ne rileva lo stato ed il processo figlio muore
7. Ritorna al punto 1

# Comandi di base

La sintassi tipica dei comandi e' la seguente:

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

## Comandi di base

- ▶ **Le opzioni** sono degli attributi opzionali che influiscono sul funzionamento del comando. Consistono generalmente di un trattino (-) seguito da una sola lettera (es. `ls -l`). Alcune opzioni 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 -laR`)

# Comandi di base

- ▶ ***Gli argomenti***, un comando puo' avere zero o piu' argomenti separati da spazi. Alcuni comandi richiedono almeno un argomento. Esistono dei comandi i cui argomenti sono opzionali, se non specificati assumono valori di default.

# Comandi di base

Esempi di comandi:

- ▶ Comandi senza argomenti:

```
pwd  
date
```

- ▶ Comandi con un solo argomento:

```
cd /tmp
```

- ▶ Comandi con una opzione ed un argomento:

```
ls -l /etc
```

- ▶ Comandi con numero arbitrario di argomenti:

```
cat file1 file2 file3
```

# Comandi Relativi al Filesystem

Comando	Descrizione
ls	Mostra i file contenuti in una directory
cd	Cambia la directory corrente
pwd	Mostra la directory corrente
cp	Copia file/directory
mv	Sposta/Rinomina file/directory
rm	Rimuove file/directory
rmdir	Rimuove directory vuote
ln	Crea hard/soft link
find	Trova file/directory
touch	Cambia l'orario di un file
chmod	Cambia i permessi di file/directory
chown	Cambia il proprietario di file/directory
chgrp	Cambia il gruppo di file/directory
df	Mostra la quantita' di spazio disponibile sul disco
du	Mostra la quantita' di spazio utilizzato sul disco
file	Mostra le proprieta' di file

# Comandi per l'editing del testo

Comando	Descrizione
echo	Stampa cio' che gli viene passato come argomento
head	Stampa le prime $n$ righe di un file
tail	Stampa le ultime $n$ righe di un file
cut	Taglia parti di file
tr	Traduce o elimina caratteri
cat	Concatena file sullo stdout
less	Mostra il contenuto di un file
more	Mostra il contenuto di un file
grep	Stampa righe corrispondenti ad un pattern
sort	Ordina un file di testo
uniq	Rimuove/Omette righe duplicate
vim	Editor di testo
sed	Stream Editor

# Comandi per la Gestione dei Processi

<b>Comando</b>	<b>Descrizione</b>
<code>ps</code>	Visualizza lo stato dei processi
<code>pstree</code>	Visualizza la gerarchia dei processi
<code>top</code>	Visualizza i processi su Linux
<code>jobs</code>	Mostra la lista dei job attivi
<code>kill</code>	Invia segnali a processi in base al PID
<code>pkill</code>	Invia segnali a processi in base nome
<code>pgrep</code>	Cerca processi tra quelli attivi
<code>nohup</code>	Esegue un processo dissociandolo dal terminale in cui viene lanciato

# Comandi vari

<b>Comando</b>	<b>Descrizione</b>
who	Mostra gli utenti loggati
date	Mostra/Cambia l'ora/data di sistema
exit	Esce dalla shell corrente
logout	Esce da una shell di login

# Controllo del terminale

Keystroke	Azione
<code>^C</code> (Ctrl-C)	Interrompe il processo in esecuzione in foreground
<code>^D</code> (Ctrl-D)	“EOF” (End Of File). Termina l’input da <code>stdin</code>
<code>^D</code> (Ctrl-D)	Esegue il logout dalla shell
<code>^L</code> (Ctrl-L)	Pulisce lo schermo (equivalente al comando <code>clear</code> )
<code>^S</code> (Ctrl-S)	Blocca lo <code>stdin</code> in un terminale
<code>^Q</code> (Ctrl-Q)	Sblocca lo <code>stdin</code> in un terminale
<code>^Z</code> (Ctrl-Z)	Sospende il processo in esecuzione in foreground

# Editing della linea di comando

<b>Keystroke</b>	<b>Azione</b>
<code>^A</code> ( <code>Ctrl-A</code> )	Sposta il cursore a inizio riga
<code>^E</code> ( <code>Ctrl-E</code> )	Sposta il cursore a fine riga
<code>^K</code> ( <code>Ctrl-K</code> )	Cancella dal carattere sotto il cursore fino al termine della riga
<code>^U</code> ( <code>Ctrl-U</code> )	Cancella dal carattere prima del il cursore fino all'inizio della riga
<code>^D</code> ( <code>Ctrl-D</code> )	Cancella il carattere sotto il cursore
<code>^W</code> ( <code>Ctrl-W</code> )	Cancella dal carattere sotto il cursore indietro fino al primo spazio
<code>^Y</code> ( <code>Ctrl-Y</code> )	Reinserisce la stringa cancellata

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

# Caratteri speciali

Alcuni caratteri hanno un significato particolare, quindi non vengono interpretati letteralmente. Alcuni caratteri speciali sono:

- ▶ il carattere spazio
- ▶ il carattere punto e virgola ;
- ▶ i caratteri per l'espansione dei nomi dei files o **Globbering**
- ▶ i caratteri di **Quoting** e di **Escape**

## Il carattere *whitespace*

Un *whitespace* (spazio bianco) consiste di uno o piu' caratteri di spazio, tabulazione o riga vuota.

Il *whitespace* separa:

- ▶ gli argomenti dei comandi
- ▶ gli argomenti delle funzioni

Le righe vuote vengono silenziosamente ignorate.

## Il carattere *whitespace*: un esempio

```
% echo Hello World  
Hello World
```

```
% echo Hello      World  
Hello World
```

```
% echo "Hello      World"  
Hello      World
```

## Il carattere “;”

```
cmd1; cmd2; cmd3; ...
```

- ▶ segnala alla shell che il comando che lo precede e' terminato
- ▶ e' possibile specificare due o piu' comandi sulla stessa riga
- ▶ se una riga contiene solo un comando e' possibile omettere “;”.

```
% echo "un comando su una riga"  
% echo "un altro comando su una riga";  
% echo "inizio listato"; ls; echo "fine listato"
```

## Filename expansion

- ▶ Bash fornisce un meccanismo di espansione dei nomi dei files, chiamato **Globbing**, che riconosce ed espande determinati caratteri, detti **Wildcards** o caratteri *jolly*.

# I Wildcard

Wildcard	Corrisponde a
?	una sola occorrenza di un solo carattere
*	zero o piu' occorrenze di caratteri
[set]	una occorrenza di un solo carattere presente in <code>set</code>
[^set]	una occorrenza di un solo carattere NON presente in <code>set</code>
[range]	una occorrenza di un solo carattere presente nel <code>range</code>
[^range]	una occorrenza di un solo rcarattere NON presente nel <code>range</code>

# I Wildcard: alcuni esempi

<b>Espressione</b>	<b>Filename corrispondenti</b>
<code>? .c</code>	1.c a.c b.c c.c ...
<code>pr*</code>	pr prova printer problema progetti.txt ...
<code>*o</code>	o oo testo ...
<code>c*o*</code>	co ciao cibo.txt ...
<code>[1-9]a</code>	1a 2a ...

## Filename expansion

```
echo *  
ls *.c  
ls [a-z]*  
ls [^1-9].c  
find . -name "*.c"  
find . -name "a*b"  
find /home/lorenzo -name "prova*"
```

## Il Quoting

- ▶ permette alla shell di interpretare caratteri che hanno un significato particolare (es. i *wildcards*) letteralmente
- ▶ Il seguente esempio non fornisce l'output che ci si potrebbe aspettare:

```
% echo *  
dir1 file1 file2
```

Infatti `*` viene interpretato come *wildcard* e non letteralmente.

## Il Quoting

Per stampare il carattere `*` lo si deve *quotare*, ossia circondare tra i caratteri:

- ▶ `"` (doppio apice)
- ▶ `'` (apice singolo)

```
% echo "*"
*
```

```
% echo '*'
```

```
*
```

# Le sequenze di *Escape*

- ▶ Le sequenze di ***Escape*** o ***Escaping***, sono una forma di quoting per un singolo carattere.
- ▶ L'escape (\) posto prima di un carattere dice alla shell di interpretarlo letteralmente.

```
% echo \\*
```

```
*
```

```
% echo \\"
```

```
"
```



# Differenza tra quoting totale e parziale

```
% echo "\ \ \ \ \ \ \ \ \"  
\\ \\
```

```
% echo '/ \ \ \ \ \ \ \ \ /'  
\\ \\ \\ \\ \\ \\ \\ \\
```

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

# I *File Descriptor*

Un ***File Descriptor*** e' una chiave astratta per accedere ad un file.

Un file descriptor e' un indice per una *entry* di una struttura dati interna al kernel che contiene i dettagli di tutti i file aperti. Questa struttura dati e' solitamente chiamata *file descriptor table*. Ogni processo possiede la propria *file descriptor table*.

Le applicazioni non potendo accedere direttamente alla *file descriptor table*, inviano questo indice al kernel attraverso una *system call*.

Ogni file, directory o device aperto ha associato un file descriptor.

# I File Descriptor

Esistono tre file descriptor standard posseduti da ogni processo:

Numero	Nome
0	stdin
1	stdout
2	stderr

Tabella: File Descriptor Standard

# I File Descriptor

- ▶ `stdin` o *Standard Input* e' un file descriptor solitamente associato alla tastiera da cui ogni processo legge l'input.
- ▶ `stdout` o *Standard Output* e' un file descriptor solitamente associato allo schermo su cui ogni processo scrive l'output.
- ▶ `stderr` o *Standard Error* e' un file descriptor solitamente associato allo schermo su cui ogni processo scrive gli errori.

# I File Descriptor

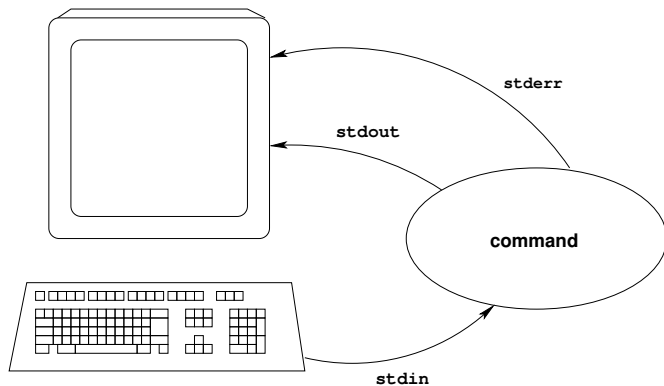


Figura: Standard File Descriptors

# Redirezione I/O

- ▶ catturare l'output e/o gli errori da un file, comando, programma o script ed inviarli come input ad un'altro file, comando, programma o script.

# Redirezione I/O

Gli operatori di redirezione sono:

<b>Operatore</b>	<b>Significato</b>
<	Redirezione dell'input
>	Redirezione dell'output
>>	Redirezione dell'output in append
2 >	Redirezione degli errori
2 >>	Redirezione degli errori in append
>&	Redirezione di un file descriptor su un'altro
	Pipe

**Tabella:** Operatori di redirezione

# Redirezione dell'input

**comando < file**

Il contenuto del file viene mandato sullo *standard input* del comando specificato. Esempio:

```
% cat < /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
bin:x:1:1:bin:/bin:/sbin/nologin  
...
```

# Redirezione dell'input

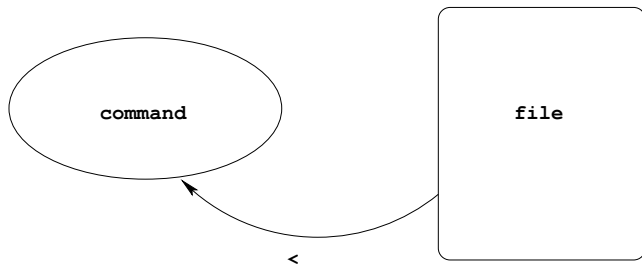


Figura: Redirezione dell'input

# Redirezione dell'input

Naturalmente il comando deve leggere da `stdin`. I comandi, infatti, possono richiedere un input in tre modi:

- ▶ da argomento di riga di comando
- ▶ da file
- ▶ da standard input

# Redirezione dell'Input

Non sono molti i comandi che leggono da standard input (`stdin`), alcuni di essi sono:

- ▶ `cat`
- ▶ `wc`
- ▶ `head`
- ▶ `tail`
- ▶ `sort`
- ▶ `uniq`
- ▶ `tr`
- ▶ `cut`
- ▶ `grep`

# Redirezione dell'Output

`comando > file`

- ▶ L'output di `comando` viene preso e rediretto sul `file`
- ▶ se il `file` non esiste viene creato.
- ▶ se il `file` esiste il contenuto del file viene sovrascritto

# Redirezione dell'Output

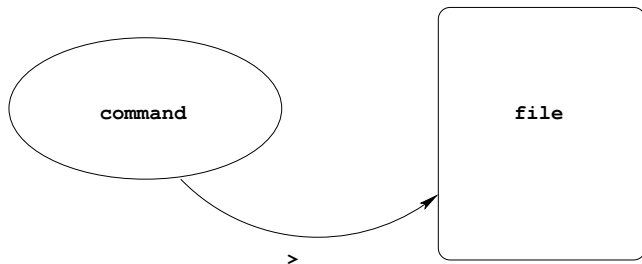


Figura: Redirezione dell'output

## Redirezione dell'Output: Esempio

```
% ls -l / > root_files
```

```
% find . -size +100M > big_files
```

```
% find . -size +300M > big_files
```

```
% ps -u lorenzo > myproc
```

# Redirezione dell'Output

**comando >> file**

L'operatore di redirezione >> e' esattamente uguale a >, con l'unica differenza che se il file esiste non viene sovrascritto ma l'output viene "accodato" al file.

```
% find . -name '*.c' > files_progetto
```

```
% find . -name '*.h' >> files_progetto
```

# Redirezione degli Errori

**comando** 2> file

Tutti gli errori vengono scritti sul file specificato. Se il file non esiste viene creato. Se il file già esiste prima dell'esecuzione del comando il contenuto del file viene sovrascritto.

Esempi:

```
% find . -name '*.sh' 2> errori
% ls 2> errori > files
% find / -name 2> /dev/null
```

# Redirezione degli Errori

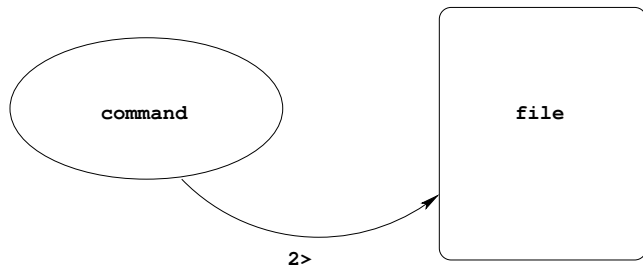


Figura: Redirezione degli errori

# Redirezione degli Errori

**comando 2>> file**

L'operatore di redirezione `2 >>` e' esattamente uguale a `2 >`, con l'unica differenza che se il file esiste non viene sovrascritto ma l'output degli errori viene "accodato" al file.

```
% find . -name '*.sh' 2>> errori
```

## Redirezione di un file descriptor su un'altro

E' possibile redirigere un file descriptor su un'altro attraverso l'operatore `>&`

**`x >& y`**

Dove `x` e' il numero del file descriptor da redirigere sul file descriptor `y`.

Esempio:

```
% find . > outfile 2>&1
```

Redirige sia lo `stdout` sia lo `stderr` sul file `outfile`.

# Redirezione e ordine degli operatori

L'ordine di esecuzione dei comandi e' da sinistra verso destra, lo stesso vale per gli operatori di redirezione:

```
% comando < infile > outfile 2>&1
```

Le operazioni vengono eseguite nel seguente ordine:

1. Viene eseguito il `comando`
2. L'origine dell'input viene cambiato da `stdin` a `infile`
3. L'output viene inviato al file `outfile` anziche' a `stdout`
4. Gli errori vengono inviati alla stessa destinazione dell'output

# Redirezione e ordine degli operatori

Il seguente comando produce un risultato inaspettato:

```
% comando < infile 2>&1 > outfile
```

Infatti gli errori vengono inviati a `stdout`, mentre l'output al file `outfile`.

# Le Pipeline

`comando1 | comando2`

- ▶ Il simbolo `|` e' chiamato *pipe*
- ▶ invia l'output di un comando come input ad un'altro comando
- ▶ Il comando a sinistra della pipe deve scrivere su `stdout`
- ▶ il comando a sinistra della pipe deve leggere da `stdin`

# Le Pipeline

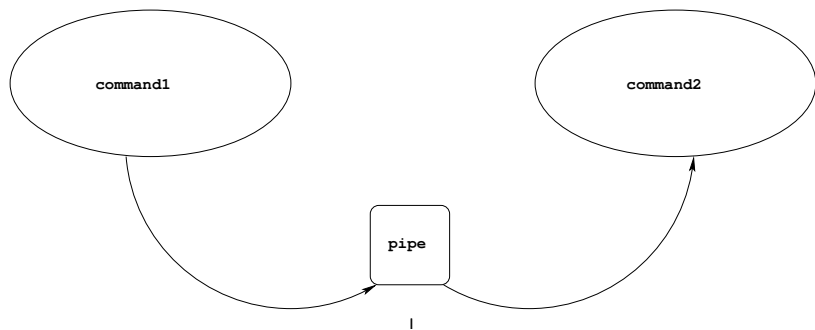


Figura: Le pipeline

# Le Pipeline

```
% ls | wc -l  
19
```

```
% cat /etc/passwd | wc -l  
39
```

E' possibile concatenare qualsiasi numero di comandi con le pipe:

```
% cat /etc/passwd | sort | uniq | tail -n 1
```

# Operatori di redirectione e pipeline

- ▶ si possono combinare: `<`, `>`, `>>`, `2>`, `2>>`, `>&` e `|`
- ▶ alcune combinazioni non hanno senso
- ▶ altre non producono i risultati che ci si aspetta

```
% tr 'a-z' 'A-Z' < file | sort > out
```

```
% find -name '*.h' 2> /dev/null | wc -l >> out
```

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

## I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

# Linguaggi di Scripting

- ▶ sono linguaggi di programmazione *interpretati* usati per scrivere programmi piu' brevi (e spesso piu' potenti) dei tradizionali linguaggi compilati.

# Linguaggi di Scripting

I linguaggi di scripting hanno varie caratteristiche:

- ▶ sono spesso implementati da un'interprete piuttosto che da un compilatore
- ▶ comunicano piu' facilmente con componenti di programmi scritti in altri linguaggi di programmazione
- ▶ l'interprete esegue direttamente le istruzioni del programma, senza alcun processo intermedio quale la compilazione.

# La Shell come linguaggio di programmazione

La Shell non e' solo un interprete di comandi, ma anche un vero e proprio linguaggio di programmazione, con variabili, funzioni e costrutti per il controllo del flusso di esecuzione del programma.

La conoscenza dello **Shell Scripting** e' essenziale per diventare un buon amministratore di sistema; infatti e' un metodo *quick and dirty* per eseguire operazioni complesse.

# Sha-Bang

Uno script e' un file di testo che deve sempre iniziare con i caratteri `#!` (*sha-bang*) che stanno ad indicare che il file contiene un'insieme di comandi da passare all'interprete specificato.

`#!` e' un ***magic number*** di due byte, che designa il tipo del file.

Dopo lo *sha-bang* si deve passare il *pathname* del programma che interpretera' i comandi contenuti nello script. Esempio:

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
```

Il filename di uno shell script ha solitamente l'estensione `".sh"`.

# Eeguire uno script

1. Uno script per essere eseguito deve avere i permessi di esecuzione:

```
% chmod +x myscript.sh
```

2. Deve essere invocato specificando:

- ▶ il pathname relativo

```
% ./myscript.sh
```

- ▶ oppure il pathname assoluto

```
% /home/lorenzo/myscript.sh
```

# I commenti

- ▶ Tutto cio' che segue il carattere # (ad eccezione dello *sha-bang* #!) e' un commento.
- ▶ I commenti vengono ignorati dalla shell, quindi un comando che segue il carattere # non viene eseguito

# I commenti

```
#!/bin/bash
# Questo e' un commento

echo "hello world" # commento

# echo "questa echo non viene eseguita"
```

## Listing 1: Esempio di Commenti

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

# Le Variabili

- ▶ Le variabili non sono altro che il modo con cui i programmi memorizzano i dati.
- ▶ Una variabile e' un'*etichetta* associata ad una o piu' locazioni di memoria.
- ▶ Diversamente da molti linguaggi di programmazione, le variabili in Bash sono prive di tipo.
- ▶ I valori delle variabili in Bash sono stringhe di caratteri che, dipendentemente dal contesto, possono essere valutate come valori numerici.
- ▶ le variabili non devono essere dichiarate, infatti esse vengono create quando vengono usate la prima volta.

# I nomi delle variabili

I nomi delle variabili devono sottostare ad alcune regole:

- ▶ Possono contenere:
  - ▶ i caratteri di lettere maiuscoli e minuscoli (`[a-zA-Z]`)
  - ▶ numeri da 0 a 9 (`[0-9]`)
  - ▶ il carattere *underscore* (`_`)
- ▶ Devono iniziare con una lettera o un carattere *underscore*

Le parentesi graffe (`{ }`) in bash servono per delimitare il nome di una variabile da altri caratteri che non fanno parte del nome stesso.

# Inizializzazione e assegnamento

## *Sintassi*

`nome_variabile=valore`

- ▶ `nome_variabile` e' il nome della variabile
- ▶ `valore` e' il valore da assegnare
- ▶ Se non esiste viene creata e gli viene assegnato il nuovo valore
- ▶ altrimenti il suo valore viene sovrascritto

# Inizializzazione e assegnamento

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

```
var1="Hello"
```

```
var2="123"
```

```
var3="Mad World"
```

Listing 2: Esempio di assegnamento di variabili

# Inizializzazione e assegnamento: ATTENZIONE!!!

- ▶ Tra l'operatore di assegnamento = e i due operandi **NON** devono essere presenti spazi o caratteri di tabulazione.

# Inizializzazione e assegnamento

Questo genera un errore:

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

```
# Errore: spazio a sinistra di =  
var1 ="prova"
```

```
# Errore: tabulazione a destra di =  
var2=  "prova2"
```

```
# Errore: spazi prima e dopo =  
var3 = "123"
```

Listing 3: Errori nell'assegnamento di variabili

# Accedere al valore delle variabili

## *Sintassi*

`$nome_variabile`

- ▶ `nome_variabile` **e'** il nome di una variabile
- ▶ `$nome_variabile` **accede** al suo valore.

# Stampare il valore delle variabili

```
#!/bin/bash

str="Una Stringa"

# Stampa: Una Stringa
echo $str

# Stampa: Una Stringa
echo "$str"

# Stampa: $str
echo '$str'

# Stampa: str
echo str

# Stampa: str
echo "str"
```

Listing 4: Esempi di stampa con `echo`

# Eliminare una variabile

## *Sintassi*

**unset** nome\_variabile

- ▶ **unset** elimina una variabile dall'environment
- ▶ **unset** imposta il valore a *null*
- ▶ e' equivalente a `nome_variabile=`

# Eliminare una variabile

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

```
x="10"
```

```
y=$x
```

```
echo "x: $x"
```

```
echo "y: $y"
```

```
unset x
```

```
echo "x: $x"
```

```
echo "y: $y"
```

Listing 5: unset.sh

# Eliminare una variabile

```
% ./unset.sh  
x: 10  
y: 10  
x:  
y: 10
```

## Variabili locali

Di default le variabili sono *locali* all'ambiente in cui vengono definite (ossia al processo in cui vengono inizializzate). Quindi non vengono ereditate dai processi figli.

# Variabili globali

## *Sintassi*

**export** [NOME [=VALORE] ]

- ▶ **export** esporta una variabile alle shell figlie della shell corrente
- ▶ Le variabili esportate sono globali solamente alle shell figlie
- ▶ Una shell figlia non puo' esportare una variabile alla shell padre
- ▶ Una shell figlia puo' essere:
  - ▶ uno script
  - ▶ una nuova istanza della shell lanciata dalla shell corrente

## Variabili Predefinite

Bash mette a disposizione un insieme di variabili predefinite, alcune di queste sono:

Variabile	Descrizione	Esempio
\$BASH	Contiene il path del binario di bash.	/bin/bash
\$EDITOR	Editor di default	vim o emacs
\$HOME	Path della home directory dell'utente	/home/lorenzo
\$HOSTNAME	Hostname della macchina	localhost
\$LOGNAME	Nome dell'utente corrente	lorenzo
\$OSTYPE	Tipo di sistema operativo	linux
\$PATH	Path delle directory per i vari programmi	/usr/bin:/bin:
\$PPID	PID del processo padre di quello corrente	3412
\$PWD	La working directory corrente	/usr
\$SHELL	Path della shell in uso	/bin/sh
\$UID	UID del proprietario del processo corrente	1000

**Tabella:** Alcune variabili d'ambiente predefinite

# Parametri Posizionali

E' possibile passare degli argomenti ad uno script, tali argomenti prendono il nome di ***parametri posizionali***, poiche' ci si fa' riferimento tramite la loro posizione sulla riga di comando (da sinistra verso destra).

# Parametri Posizionali

Tali argomenti vengono associati a delle variabili in modo tale da poter essere gestiti all'interno dello script:

- ▶ `$0` e' il nome dello script
- ▶ `$1` – `$9` sono gli argomenti dello script
  - ▶ `$1` e' il primo argomento
  - ▶ `$2` e' il secondo argomento
  - ▶ ...
  - ▶ `$9` e' il nono argomento
- ▶ `$#` contiene il numero di parametri passati allo script
- ▶ `$*` contiene tutti i parametri posizionali in una sola parola

# Parametri Posizionali

```
#!/bin/bash  
  
echo "Numero parametri \ $#: $#"  
echo "Lista parametri \ $*: $*"  
echo "Nome script \ $0: $0"  
echo "Parametro \ $1: $1"  
echo "Parametro \ $2: $2"  
echo "Parametro \ $3: $3"
```

Listing 6: param.sh

# Exit status

Quando un processo termina, ritorna un ***exit status*** al processo padre. L'exit status e' un numero che indica se l'esecuzione del programma e' andata a buon fine o meno.

Per convenzione:

- ▶ *il valore 0* indica che l'esecuzione del programma e' andata a buon fine
- ▶ *Un valore diverso da 0* indica che c'e' stato un problema.

# Exit status

```
% ls  
% echo $?  
0
```

```
% ls blablabla  
ls: blablabla: No such file or directory  
% echo $?  
2
```

```
% grep "root" /etc/passwd  
root:x:0:0:root:/root:/bin/bas  
% echo $?  
0
```

```
% grep "blablabla" /etc/passwd  
% echo $?  
1
```

## Sostituzione dei comandi

La **sostituzione di un comando** riassegna l'output di un comando a un'altro comando, a una variabile, a un costrutto.

La forma classica di sostituzione di un comando si effettua circondando il comando da sostituire tra *backquotes* ``...`` (o *backticks*), oppure con il costrutto `$( )`:

```
`comando`
```

Oppure

```
$(comando)
```

```
% echo $(ls)  
dir1 dir2 file1 file2 prova.txt  
% rm `cat file_da_rimuovere`  
% files=$(ls *.txt)  
% echo $files  
prova.txt
```

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

### Flusso di Esecuzione

if

for

while

case

# Espansione Aritmetica

L'espansione aritmetica permette di valutare un'espressione aritmetica e sostituirvi il risultato.

La sintassi e' la seguente:

```
$( (espressione) )
```

Esempio:

```
% echo $( (5 + 5) )
```

```
10
```

```
% var=$( (1+1) )
```

```
% echo $( ($var-1) )
```

```
1
```

```
% var=$( ($var+1) )
```

# Il comando `expr`

## *Sintassi*

**expr** ESPRESSIONE

- ▶ **expr** valuta ESPRESSIONE e stampa il risultato sullo `stdout`
- ▶ ESPRESSIONE puo' contenere valori numerici, operatori aritmetici e parentesi tonde "quotate" `\( e \)`
- ▶ Tra gli operatori e gli operandi dev'essere presente uno spazio

## Il comando `expr` (2)

```
% expr 5 + 5  
10  
% expr 4 \* 2  
8  
% expr 4 + 2 \* 3  
10  
% expr \( 4 + 2 \) \* 3  
18  
% var=$(expr 3 + 3)
```

# Il comando `let`

## *Sintassi*

**let** ESPRESSIONE

- ▶ Valuta operazioni aritmetiche senza produrre alcun output.
- ▶ Essendo un comando *builtin*, permette di memorizzare il risultato di una espressione in una variabile:

## Il comando `let` (2)

```
% let "x=5+5"  
% echo $x  
10  
% let "y=$x*2"  
% echo $y  
20  
% let "y++"  
% echo $y  
21
```

# Outline

## Introduzione

Storia

Invocare la shell

I Comandi

I caratteri di controllo

## Caratteri Speciali

Filename Expansion

Quoting

## Redirezione

Redirezione dell'input

Redirezione dell'output

Redirezione degli errori

Le Pipeline

## Shell Scripting

I Commenti

## Le Variabili

Assegnamento

Valore delle variabili

Eliminare una variabile

Parametri posizionali

Exit Status

Sostituzione dei comandi

## Operazioni Aritmetiche

## Flusso di Esecuzione

if

for

while

case

## Il costrutto `if`

- ▶ E' un costrutto condizionale, in base ad una condizione sceglie se eseguire un blocco di istruzioni oppure no
- ▶ Il costrutto `if` esegue un blocco di istruzioni solo se la *condizione e' vera*.

## Il costrutto `if` (2)

### *Sintassi*

```
if comando  
then  
    statements  
    ...  
fi
```

La *condizione* controllata dal costrutto `if` e' l'exit status del comando:

- ▶ se vale 0 allora la condizione viene valutata come vera e vengono eseguite le istruzioni.
- ▶ per qualsiasi altro valore la condizione e' considerata falsa.

## Il costrutto `if` (3)

```
if grep "$var" /etc/passwd  
then  
    echo "Utente $var trovato"  
fi
```

# Il costrutto `if`: la clausola `else`

## *Sintassi*

```
if comando1
then
    statement1
    ...
else
    statement2
    ...
fi
```

- ▶ Solo un blocco di istruzioni puo' essere eseguito
- ▶ Sempre in base all'exit status del *comando*
- ▶ Eseguito uno dei due blocchi, l'esecuzione riprende dalla prima istruzione dopo il **fi**

## Il costrutto `if`: la clausola `else`

```
if grep "$var" /etc/passwd
then
    echo "Utente $var trovato"
else
    echo "Utente $var non trovato"
fi
```

# Il costrutto `if`: la clausola `elif`

## *Sintassi*

```
if comando1
then
    statement1
    ...
elif comando2
then
    statement2
    ...
elif comando3
then
    statement3
    ...
else
    statement4
    ...
fi
```

## Il costrutto `if`: la clausola `elif` (2)

```
if grep "$var" /etc/passwd
then
    echo "Utente $var trovato"

elif grep "$var" /etc/group
then
    echo "Gruppo $var trovato"

else
    echo "Utente/Gruppo $var non trovato"
fi
```

## Il costrutto `if` e la redirectione

```
if grep "$var" /etc/passwd > /dev/null 2>&1
then
    echo "Utente $var trovato"

elif grep "$var" /etc/group > /dev/null 2>&1
then
    echo "Gruppo $var trovato"

else
    echo "Utente/Gruppo $var non trovato"
fi
```

## Il costrutto `if` e le pipe

- ▶ E' possibile concatenare piu' comandi con le pipe nelle clausole `if` e `elif`
- ▶ In questo caso l'*exit status* valutato e' quello dell'ultimo comando a destra della pipe

```
if ps aux | cut -f 1 -d ' ' | grep "$user"
then
    echo "$user sta eseguendo uno o piu' processi"
else
    echo "$user non sta eseguendo alcun processo"
fi
```

# “Testare” le Condizioni

## *Sintassi*

**test** [ESPRESSIONE]

- ▶ E' un comando *builtin*
- ▶ Ritorna come exit status:
  - ▶ 0 se il risultato dell'ESPRESSIONE e' vero
  - ▶ 1 se il risultato dell'ESPRESSIONE e' falso
- ▶ ESPRESSIONE puo' essere:
  - ▶ una costante
  - ▶ il valore di una variabile
  - ▶ un file

## Il comando `test`

`test` si puo' usare nel costrutto `if` nei seguenti modi:

```
if test ESPRESSIONE
then
    statements
fi
```

Oppure con le parentesi quadre:

```
if [ ESPRESSIONE ]
then
    statements
fi
```

Ed infine il metodo consigliato (solo su Bash2):

```
if [[ ESPRESSIONE ]]
then
    statements
fi
```

# Gli operatori di `test`

L'ESPRESSIONE puo' contenere determinati operatori che vengono interpretati da `test`, tali operatori si dividono in:

- ▶ Operatori relazionali numerici
- ▶ Operatori relazionali per le stringhe
- ▶ Operatori per i file

# Operatori relazionali numerici di `test`

Operatore	Esempi	Significato
<code>-eq</code>	<code>test "\$a" -eq "\$b"</code>	Gli operandi sono uguali
<code>-ne</code>	<code>test "\$a" -ne "\$b"</code>	Gli operandi sono diversi
<code>-gt</code>	<code>test "\$a" -gt "\$b"</code>	<code>\$a</code> e' maggiore di <code>\$b</code>
<code>-ge</code>	<code>test "\$a" -ge "\$b"</code>	<code>\$a</code> e' maggiore uguale di <code>\$b</code>
<code>-lt</code>	<code>test "\$a" -lt "\$b"</code>	<code>\$a</code> e' minore di <code>\$b</code>
<code>-le</code>	<code>test "\$a" -le "\$b"</code>	<code>\$a</code> e' minore uguale di <code>\$b</code>

Tabella: Operatori relazionali numerici di `test`

## Operatori relazionali per le stringhe di `test`

Operatore	Esempi	Significato
<code>=</code>	<code>test "\$str1" = "\$str2"</code>	Gli operandi sono uguali
<code>!=</code>	<code>test "\$str1" != "\$str2"</code>	Gli operandi sono diversi
<code>&gt;</code>	<code>test "\$str1" &gt; "\$str2"</code>	<code>\$str1</code> e' maggiore di <code>\$str2</code>
<code>&gt;=</code>	<code>test "\$str1" &gt;= "\$str2"</code>	<code>\$str1</code> e' maggiore uguale di <code>\$str2</code>
<code>&lt;</code>	<code>test "\$str1" &lt; "\$str2"</code>	<code>\$str1</code> e' minore di <code>\$str2</code>
<code>&lt;=</code>	<code>test "\$str1" &lt;= "\$str2"</code>	<code>\$str1</code> e' minore uguale di <code>\$str2</code>
<code>-z</code>	<code>test -z "\$str1"</code>	<code>\$str1</code> e' nulla (ha lunghezza 0)
<code>-n</code>	<code>test -n "\$str1"</code>	<code>\$str1</code> e' non e' nulla

**Tabella:** Operatori di `test` per le stringhe

## Operatori per i file di `test`

Gli operatori per i file di `test` controllano alcune condizioni sui file, alcuni di essi sono:

Operatore	Esempi	Significato
<code>-e</code>	<code>test -e path</code>	Il file <code>path</code> esiste
<code>-f</code>	<code>test -f path</code>	Il file <code>path</code> esiste ed e' regolare
<code>-d</code>	<code>test -d path</code>	Il file <code>path</code> esiste ed e' una directory
<code>-s</code>	<code>test -s path</code>	Il file <code>path</code> ha grandezza > 0
<code>-r</code>	<code>test -r path</code>	Il file <code>path</code> ha permessi di lettura
<code>-w</code>	<code>test -w path</code>	Il file <code>path</code> ha permessi di scrittura
<code>-x</code>	<code>test -x path</code>	Il file <code>path</code> ha permessi di esecuzione

**Tabella:** Operatori di `test` per i file

# Il costrutto `for`

## *Sintassi*

```
for VARIABILE in LISTA  
do  
    STATEMENTS  
done
```

- ▶ `VARIABILE` e' il nome di una variabile che ad ogni iterazione assume un valore della `LISTA`
- ▶ `LISTA` e' una lista di stringhe separate da spazi

## Il costrutto `for`: Esempio

```
for i in qui quo qua
do
    echo $i
done
```

## Il costrutto `for` e la sostituzione dei comandi

Si puo' usare la sostituzione dei comandi per generare una lista:

```
for file in $(ls *.c)
do
    rm -f $file
    echo "$file eliminato"
done
```

## Il costrutto `for` e le pipe

Per elencare gli utenti presenti sul sistema:

```
n=1
for user in $(cut -d : -f1 /etc/passwd | sort)
do
    echo "Utente $n: $user"
    let "n++"
done
```

## Il costrutto `for`

Stampare la dimensione dei file nella directory corrente:

```
for file in *
do
    if [[ -f $file ]]
    then
        du -h $file
    fi
done
```

Versione migliorata:

```
lista=$(ls | tr ' ' '%')
```

```
for file in $lista
do
    file=$(echo $file | tr '% ' ' ')
    if [[ -f "$file" ]]
    then
        du -h "$file"
    fi
done
```

## Il costrutto `for`

Per eseguire una operazione  $n$  volte si puo' utilizzare il comando `seq`:

```
for i in $(seq 1 10)
do
    echo "Iterazione $i"
done
```

Oppure, poiche' `seq` non si trova su tutti i sistemi, si usa la seguente forma "*C-Style*" del costrutto `for`:

```
for ((i=1;i<= 10;i++))
do
    echo "Iterazione $i"
done
```

# Il costrutto `while`

## *Sintassi*

```
while COMANDO  
do  
    STATEMENTS  
done
```

- ▶ Il costrutto **while** esegue gli `STATEMENTS` finché `COMANDO` ha come exit status 0
- ▶ Come `COMANDO` si può usare `test` come nel costrutto **if**

## Il costrutto while

```
i=0
while [[ "$i" -ne "10" ]]
do
    echo -n "$i "
    i=$(expr $i + 1)
done
```

```
while who | grep "lorenzo" > /dev/null
do
    echo "lorenzo is logged in!"
done
```

# Il costrutto `until`

## *Sintassi*

```
until COMANDO  
do  
    STATEMENTS  
done
```

- ▶ Il costrutto `until` esegue gli `STATEMENTS` finché `COMANDO` ha un exit status *diverso* da 0
- ▶ Come `COMANDO` si può usare `test` come nel costrutto `if`

# Il costrutto `until`

```
i=1  
  
until (( $i > 10 ))  
do  
    echo -n "$i "  
    let "i+=1"  
done
```

# I cicli infiniti

- ▶ Un ciclo infinito e' un ciclo che non termina mai a meno che non venga espressamente interrotto con il comando **break**.
- ▶ Per realizzare un ciclo infinito con il costrutto **while** e' sufficiente utilizzare nell'espressione un comando che termini sempre con successo.
- ▶ Il comando **true** fa' al caso nostro: **true** non fa' nient'altro che ritornare un exit status pari a zero:

```
while true
do
    echo "loop"
done
```

## I cicli infiniti (2)

Il comando complementare a `true` e' `false`, che ritorna sempre 1 come exit status. Quindi un loop come il seguente non eseguirà mai alcuna iterazione:

```
while false
do
    echo "loop"
done
```

`false` ha la sua utilità se utilizzato con il costrutto `until` per creare un loop infinito:

```
until false
do
    echo "loop"
done
```

# Il costrutto case

## *Sintassi*

```
case ESPRESSIONE in
```

```
    PATTERN1)  
        STATEMENTS1  
    ;;
```

```
    PATTERN2)  
        STATEMENTS2  
    ;;
```

```
    . . . .
```

```
esac
```

## Il costrutto `case` (2)

- ▶ Controlla il valore di `ESPRESSIONE` se corrisponde ad uno dei `pattern`, vengono eseguiti gli *statement* corrispondenti
- ▶ I *pattern* sono stringhe che possono contenere i caratteri *wildcard*
- ▶ Come nei linguaggi convenzionali, `case` corrisponde ad una serie di `statement if-then-else`
- ▶ Se nessun `pattern` corrisponde allora non viene eseguito alcun `statement`
- ▶ E' possibile specificare piu' `pattern` separati dal carattere pipe `|`
- ▶ E' possibile annidare due o piu' costrutti `case`

## Il costrutto case (3)

```
case $var in
```

```
1)
```

```
    echo "\$var vale 1"  
    ;;
```

```
2)
```

```
    echo "\$var vale 2"  
    ;;
```

```
esac
```

## Il costrutto case (4)

```
case $filename in
  *.gif)
    echo "GIF Format"
    ;;
  *.png)
    echo "PNG Format"
    ;;
  *.jpg | *.jpeg)
    echo "JPEG Format"
    ;;
  *)
    echo "Unknown image format"
    ;;
esac
```