Corso facilitato di bash scripting

By CasertaGLUG per informazioni contattare l'autore casertaglug-owner@autistici.org

Alcune Parti del manuale contengono contenuti dell'originale più indispensabili.

Parte 2 (Collegamenti alla guida completa)

Caratteri Speciali

(#) Con questo simbolo iniziate un commento che viene automaticamente ignorato dall'interprete. Può essere posto anche affianco ad un altro comando. Inoltre può essere preceduto anche da un carattere di tabulazione (es 1:[tabul]# tuo commento).

(#!) Questo con il punto esclamativo a differenza del precedente serve a specificare un percorso di dipendenze (Vedi Corso Prima Parte rif #!) (con questa stringa di percorso (/bin/bash) se dobbiamo programmare in bash.

(echo) Serve a visualizzare il testo nel terminale.

Non è possibile inserire, sulla stessa riga, un comando dopo un commento. Non esiste alcun metodo per terminare un commento in modo che si possa inserire del "codice", eseguibile, sulla stessa riga. È indispensabile porre il comando in una nuova riga.

Funzionalità complesse spiegate nella guida originale riportate di seguito:

`${PATH#*:}`

È una sostituzione di parametro, non un commento. Si inserisce nel comando echo in sostituzione dell'inserimento di un testo da visualizzare.

$((2#101011))
È una conversione di base, non un commento. Convertitore (Si vedano gli aspetti più semplici

Cos'è il **quiting e l'escaping**?

Con il termine "quoting" si intende semplicemente questo: inserire una stringa tra apici. Viene utilizzato per proteggere i caratteri speciali contenuti nella stringa dalla reinterpretazione o espansione da parte della shell o di uno script. (Un carattere si definisce "speciale" se viene interpretato diversamente dal suo significato letterale, come il carattere jolly *.)

Esempio si veda testo con caratteri speciali.

```
bash$ ls -l [Vv]*
 -rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

Separatore di comandi [punto e virgola]. Permette di inserire due o più comandi sulla stessa riga.

```
  1 echo ehilà; echo ciao
  2
  3
  4 if [ -x "$nomefile" ]; then    #  Notate che "if" e "then" hanno
bisogno del
  5                                #+ punto e virgola. Perchè?
  6   echo "Il file $nomefile esiste."; cp $nomefile $nomefile.bak
  7 else
  8   echo "$nomefile non trovato."; touch $nomefile
  9 fi; echo "Verifica di file completata."
```

Delimitatore in un'opzione case [doppio punto e virgola].

```
  1 case "$variabile" in
  2 abc)  echo "\$variabile = abc" ;;
  3 xyz)  echo "\$variabile = xyz" ;;
  4 esac
```

.

Comando "punto" [punto]. Equivale a source

"punto", componente di nomi di file. Quando si ha a che fare con i nomi di file si deve sapere che il punto è il prefisso dei file "nascosti", file che un normale comando ls non visualizza.

```
bash$ touch .file_nascosto
bash$ ls -l
total 10
```

```
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook
employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000
employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .file_nascosto
```

Se si considerano i nomi delle directory, *un punto singolo* rappresenta la directory di lavoro corrente, mentre *due punti* indicano la directory superiore.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

Il *punto* appare spesso come destinazione (directory) nei comandi di spostamento di file.

```
bash$ cp /home/bozo/current_work/junk/* .
```

.

"punto" corrispondenza di carattere. Nella ricerca di caratteri, come parte di una espressione regolare, il "punto" verifica un singolo carattere.

"

quoting parziale [doppio apice]. *"STRINGA"* preserva (dall'interpretazione della shell) la maggior parte dei caratteri speciali che dovessero trovarsi all'interno di *STRINGA*. Vedi anche Capitolo 5.

'

quoting totale [apice singolo]. *'STRINGA'* preserva (dall'interpretazione della shell) tutti i caratteri speciali che dovessero trovarsi all'interno di *STRINGA*. Questa è una forma di quoting più forte di ". Vedi anche Capitolo 5.

,

operatore virgola. L'**operatore virgola** concatena una serie di operazioni aritmetiche. Vengono valutate tutte, ma viene restituita solo l'ultima.

```
1 let "t2 = ((a = 9, 15 / 3))"  # Imposta "a" e calcola "t2".
```

\

escape [barra inversa]. Strumento per il quoting di caratteri singoli.

\X "preserva" il carattere *X*. Equivale ad effettuare il "quoting" di *X*, vale a dire *'X'*. La \ si utilizza per il quoting di " e ', affinché siano interpretati letteralmente.

Vedi Capitolo 5 per una spiegazione approfondita dei caratteri di escape.

/

Separatore nel percorso dei file [barra]. Separa i componenti del nome del file (come in `/home/bozo/projects/Makefile`).

È anche l'operatore aritmetico di divisione.

`

sostituzione di comando. Il costrutto *`comando`* rende disponibile l'output di *comando* per impostare una variabile. È conosciuto anche come apostrofo inverso o apice inverso.

:

comando null [due punti]. È l'equivalente shell di "NOP" (*no op*, operazione non-far-niente). Può essere considerato un sinonimo del builtin di shell true. Il comando ":" è esso stesso un builtin Bash, ed il suo exit status è "true" (0).

```
1 :
2 echo $?   # 0
```

Ciclo infinito:

```
 1 while :
 2 do
 3    operazione-1
 4    operazione-2
 5    ...
 6    operazione-n
 7 done
 8
 9 # Uguale a:
10 #    while true
11 #    do
12 #      ...
13 #    done
```

Istruzione nulla in un costrutto if/then:

```
  1 if condizione
  2 then :   # Non fa niente e salta alla prossima istruzione
  3 else
  4    fa-qualcosa
  5 fi
```

Fornisce un segnaposto dove è attesa un'operazione binaria, vedi Esempio 8-2 e parametri predefiniti.

```
  1 : ${nomeutente=`whoami`}
  2 # ${nomeutente=`whoami`}   senza i : iniziali dà un errore,
  3 #                          tranne se "nomeutente" è un comando o un
builtin ...
```

Fornisce un segnaposto dove è atteso un comando in un here document. Vedi Esempio 17-10.

Valuta una stringa di variabili utilizzando la sostituzione di parametro (come in Esempio 9-13).

```
  1 : ${HOSTNAME?} ${USER?} ${MAIL?}
  2 #  Visualizza un messaggio d'errore se una, o più, delle variabili
  3 #+ fondamentali d'ambiente non è impostata.
```

Espansione di variabile / sostituzione di sottostringa.

In combinazione con >, l'operatore di redirezione, azzera il contenuto di un file, senza cambiarne i permessi. Se il file non esiste, viene creato.

```
  1 : > data.xxx   # Ora il file "data.xxx" è vuoto.
  2
  3 # Ha lo stesso effetto di cat /dev/null > data.xxx
  4 # Tuttavia non viene generato un nuovo processo poiché ":" è un
builtin.
```

Si faccia attenzione che ";", talvolta, deve essere preceduto da un carattere di escape

In combinazione con l'operatore di redirezione >> non ha alcun effetto su un preesistente file di riferimento (`: >> file_di_riferimento`). Se il file non esiste, viene creato.


Si utilizza solo con i file regolari, non con le pipe, i link simbolici ed alcuni file particolari.

In questa sezione del manuale originale indispensabile, dove non conviene riassumere certi termini, verranno riportati esattamente come sono.

(":") servono anche come separatore di campo nel file `/etc/passwd` e nella variabile $PATH.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

inverte (o nega) il senso di una verifica o di un exit status. L'operatore ! inverte l'exit status di un comando a cui è stato anteposto. Cambia anche il significato di un operatore di verifica. Può, per esempio, cambiare il senso di "uguale" (=) in "non uguale" (!=). L'operatore ! è una parola chiave Bash.

In un contesto differente, il ! appare anche nelle referenziazioni indirette di variabili.

Ancora, da *riga di comando* il ! invoca il *meccanismo della cronologia* di Bash (vedi Appendice I). È da notare che, all'interno di uno script, il meccanismo della cronologia è disabilitato.

carattere jolly [asterisco]. Il carattere * serve da "carattere jolly" per l'espansione dei nomi di file nel globbing. Da solo, ricerca tutti i file di una data directory.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

L' * rappresenta anche tutti i caratteri (o nessuno) in una espressione regolare.

*

operatore aritmetico. Nell'ambito delle operazioni aritmetiche, l' * indica l'operatore di moltiplicazione.

Il doppio asterisco, **, è l'operatore di elevamento a potenza.

?

operatore di verifica. In certe espressioni, il ? indica la verifica di una condizione.

In un costrutto parentesi doppia, il ? viene utilizzato come operatore ternario in stile C. Vedi Esempio 9-29.

Nella sostituzione di parametro, il ? verifica se una variabile è stata impostata.

?

carattere jolly. Il carattere ? serve da "carattere jolly" per un singolo carattere, nell'espansione dei nomi di file nel globbing, così come rappresenta un singolo carattere in una espressione regolare estesa.

$

Sostituzione di variabile.

```
1 var1=5
2 var2=23skidoo
3
```

```
4 echo $var1     # 5
5 echo $var2     # 23skidoo
```

Il $ davanti al nome di una variabile indica il *valore* contenuto nella variabile stessa.

$

fine-riga. In una espressione regolare, il "$" rinvia alla fine della riga di testo.

${}

Sostituzione di parametro.

$*, $@

Parametri posizionali.

$?

variabile exit status. La variabile $? contiene l'exit status di un comando, di una funzione, o dello stesso script.

$$

variabile ID di processo. La variabile $$ contiene l'*ID di processo* dello script in cui appare.

()

gruppo di comandi.

```
1 (a=ciao; echo $a)
```

Un elenco di comandi racchiusi da *`parentesi`* dà luogo ad una subshell.

Le variabili all'interno delle parentesi, appartenenti quindi alla subshell, non sono visibili dallo script. Il processo genitore, lo script, non può leggere le variabili create nel processo figlio, la subshell.

```
1 a=123
2 ( a=321; )
3
4 echo "a = $a"   # a = 123
5 # "a" tra parentesi si comporta come una variabile
locale.
```

inizializzazione di array.

```
1 Array=(elemento1 elemento2 elemento3)
```

{xxx,yyy,zzz,...}

Espansione multipla.

```
  1 grep Linux file*.{txt,htm*}
  2 # Cerca tutte le ricorrenze della parola "Linux"
  3 # nei file "fileA.txt", "file2.txt", "fileR.html", "file-87.htm",
etc.
```

Il comando agisce sull'elenco dei file, separati da virgole, specificati tra le *parentesi graffe*. [1] L'espansione dei nomi dei file (il globbing) viene applicata a quelli elencati tra le parentesi.

Non è consentito alcuno spazio dentro le parentesi, *tranne il caso* in cui si utilizzi il "quoting" o se preceduto da un carattere di escape.

```
echo {file1,file2}\ :{\ A," B",' C'}

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

{}

Blocco di codice [parentesi graffe]. Conosciuto anche come "gruppo inline", questo costrutto crea una funzione anonima. Tuttavia, a differenza di una funzione, le variabili presenti nel blocco rimangono visibili alla parte restante dello script.

```
bash$ { local a;
              a=123; }
bash: local: can only be used in a
function
```

```
  1 a=123
  2 { a=321; }
  3 echo "a = $a"   # a = 321   (valore di a nel blocco di codice)
  4
  5 # Grazie, S.C.
```

La porzione di codice racchiusa tra le parentesi graffe può avere l'I/O rediretto da e verso se stessa.

Esempio 3-1. Blocchi di codice e redirezione I/O

```
  1 #!/bin/bash
  2 # Legge le righe del file /etc/fstab.
  3
  4 File=/etc/fstab
  5
  6 {
  7 read riga1
  8 read riga2
  9 } < $File
 10
 11 echo "La prima riga di $File è:"
 12 echo "$riga1"
 13 echo
 14 echo "La seconda riga di $File è:"
 15 echo "$riga2"
 16
 17 exit 0
```

Esempio 3-2. Salvare i risultati di un blocco di codice in un file

```
 1 #!/bin/bash
 2 # rpm-check.sh
 3
 4 # Interroga un file rpm per visualizzarne la descrizione ed il
 5 #+contenuto, verifica anche se può essere installato.
 6 # Salva l'output in un file.
 7 #
 8 # Lo script illustra l'utilizzo del blocco di codice.
 9
10 SUCCESSO=0
11 E_ERR_ARG=65
12
13 if [ -z "$1" ]
14 then
15   echo "Utilizzo: `basename $0` file-rpm"
16   exit $E_ERR_ARG
17 fi
18
19 {
20   echo
21   echo "Descrizione Archivio:"
22   rpm -qpi $1       # Richiede la descrizione.
23   echo
24   echo "Contenuto dell'archivio:"
25   rpm -qpl $1       # Richiede il contenuto.
26   echo
27   rpm -i --test $1  # Verifica se il file rpm può essere installato.
28   if [ "$?" -eq $SUCCESSO ]
29   then
30     echo "$1 può essere installato."
31   else
32     echo "$1 non può essere installato."
33   fi
34   echo
35 } > "$1.test"       #  Redirige l'output di tutte le istruzioni del
blocco
36                     #+ in un file.
37
38 echo "I risultati della verifica rpm si trovano nel file $1.test"
39
40 # Vedere la pagina di manuale di rpm per la spiegazione delle
opzioni.
41
42 exit 0
```

A differenza di un gruppo di comandi racchiuso da (parentesi), visto in precedenza, una porzione di codice all'interno di {parentesi graffe} solitamente *non* dà vita ad una subshell. [2]

{} \;

percorso del file. Per lo più utilizzata nei costrutti find. *Non* è un builtin di shell.

Il ";" termina la sequenza dell'opzione `-exec` del comando **find**. Deve essere preceduto dal carattere di escape per impedirne la reinterpretazione da parte della shell.

[ ]

verifica.

Verifica l'espressione tra **[ ]**. È da notare che **[** è parte del builtin di shell **test** (ed anche suo sinonimo), *non* un link al comando esterno `/usr/bin/test`.

[[ ]]

verifica.

Verifica l'espressione tra [[ ]] (parola chiave di shell).

Vedi la disamina sul costrutto [[...]].

[ ]

elemento di un array.

Nell'ambito degli array, le parentesi quadre vengono impiegate nell'impostazione dei singoli elementi di quell'array.

```
1 Array[1]=slot_1
2 echo ${Array[1]}
```

[ ]

intervallo di caratteri.

Come parte di un'espressione regolare, le parentesi quadre indicano un intervallo di caratteri da ricercare.

(( ))

espansione di espressioni intere.

Espande e valuta l'espressione intera tra (( )).

Vedi la disamina sul costrutto ((...)).

> &> >&>> <

redirezione.

`nome_script >nome_file` redirige l'output di `nome_script` nel file `nome_file`. Sovrascrive `nome_file` nel caso fosse già esistente.

`comando &>nome_file` redirige sia lo `stdout` che lo `stderr` di `comando` in `nome_file`.

`comando >&2` redirige lo `stdout` di `comando` nello `stderr`.

`nome_script >>nome_file` accoda l'output di `nome_script` in `nome_file`. Se `nome_file` non esiste, viene creato.

sostituzione di processo.

(comando)>

<(comando)

In un altro ambito, i caratteri "<" e ">" vengono utilizzati come operatori di confronto tra stringhe.

In un altro ambito ancora, i caratteri "<" e ">" vengono utilizzati come operatori di confronto tra interi. Vedi anche Esempio 12-9.

<<

redirezione utilizzata in un here document.

<<<

redirezione utilizzata in una here string.

<, >

Confronto ASCII.

```
 1 veg1=carote
 2 veg2=pomodori
 3
 4 if [[ "$veg1" < "$veg2" ]]
 5 then
 6   echo "Sebbene nel dizionario $veg1 preceda $veg2,"
 7   echo "questo non intacca le mie preferenze culinarie."
 8 else
 9   echo "Che razza di dizionario stai usando?"
10 fi
```

\<, \>

delimitatore di parole in un'espressione regolare.

```
bash$ grep '\<il\>' filetesto
```

|

pipe. Passa l'output del comando che la precede come input del comando che la segue, o alla shell. È il metodo per concatenare comandi.

```
 1 echo ls -l | sh
 2 #  Passa l'output di "echo ls -l" alla shell,
 3 #+ con lo stesso risultato di "ls -l".
 4
 5
 6 cat *.lst | sort | uniq
 7 # Unisce ed ordina tutti i file ".lst", dopo di che cancella le
righe doppie.
```

Una pipe, metodo classico della comunicazione tra processi, invia lo `stdout` di un processo allo `stdin` di un altro. Nel caso tipico di un comando, come cat o echo, collega un flusso di dati da elaborare ad un "filtro" (un comando che trasforma il suo input).

```
cat $nome_file1 $nome_file2 | grep $parola_da_cercare
```

L'output di uno o più comandi può essere collegato con una pipe ad uno script.

```
  1 #!/bin/bash
  2 # uppercase.sh : Cambia l'input in caratteri maiuscoli.
  3
  4 tr 'a-z' 'A-Z'
  5 #  Per l'intervallo delle lettere deve essere utilizzato il
"quoting" per
  6 #+ impedire di creare file aventi per nome le singole lettere dei
nomi
  7 #+ dei file.
  8
  9 exit 0
```

Ora si collega l'output di **ls -l** allo script.

```
bash$ ls -l | ./uppercase.sh
 -RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

In una pipe, lo `stdout` di ogni processo deve essere letto come `stdin` del successivo. Se questo non avviene, il flusso di dati si *blocca*. La pipe non si comporterà come ci si poteva aspettare.

```
  1 cat file1 file2 | ls -l | sort
  2 # L'output proveniente da "cat file1 file2" scompare.
```

Una pipe viene eseguita come processo figlio e quindi non può modificare le variabili dello script.

```
  1 variabile="valore_iniziale"
  2 echo "nuovo_valore" | read variabile
  3 echo "variabile = $variabile"     # variabile =
valore_iniziale
```

Se uno dei comandi della pipe abortisce, questo ne determina l'interruzione prematura. Chiamata *pipe troncata*, questa condizione invia un segnale *SIGPIPE*.

>|

forza la redirezione (anche se è stata impostata l'opzione noclobber) . Ciò provoca la sovrascrittura forzata di un file esistente.

||

operatore logico OR. In un costrutto condizionale, l'operatore || restituirà 0 (successo) se *almeno una* delle condizioni di verifica valutate è vera.

&

Esegue un lavoro in background. Un comando seguito da una & verrà eseguito in background (sullo sfondo).

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

In uno script possono essere eseguiti in background sia i comandi che i cicli .

Esempio 3-3. Eseguire un ciclo in background

```
 1 #!/bin/bash
 2 # background-loop.sh
 3
 4 for i in 1 2 3 4 5 6 7 8 9 10            # Primo ciclo.
 5 do
 6   echo -n "$i "
 7 done & # Esegue questo ciclo in background.
 8        # Talvolta verrà eseguito, invece, il secondo ciclo.
 9
10 echo   # Questo 'echo' alcune volte non verrà eseguito.
11
12 for i in 11 12 13 14 15 16 17 18 19 20   # Secondo ciclo.
13 do
14   echo -n "$i "
15 done
16
17 echo   # Questo 'echo' alcune volte non verrà eseguito.
18
19 # ======================================================
20
21 # Output atteso:
22 # 1 2 3 4 5 6 7 8 9 10
23 # 11 12 13 14 15 16 17 18 19 20
24
25 # Talvolta si potrebbe ottenere:
26 # 11 12 13 14 15 16 17 18 19 20
27 # 1 2 3 4 5 6 7 8 9 10 bozo $
28 # (Il secondo 'echo' non è stato eseguito. Perché?)
29
30 # Occasionalmente anche:
31 # 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32 # (Il primo 'echo' non è stato eseguito. Perché?)
33
34 #  Molto raramente qualcosa come:
35 #  11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
36 #  Il ciclo in primo piano (foreground) ha la precedenza su
37 #+ quello in background.
38
39 exit 0
```

Un comando eseguito in background all'interno di uno script può provocarne l'interruzione, in attesa che venga premuto un tasto. Fortunatamente, per questa

eventualità c'è un rimedio.

&&

operatore logico AND . In un costrutto condizionale, l'operatore && restituirà 0 (successo) solo se *tutte* le condizioni verificate sono vere.

-

opzione, prefisso. Prefisso di opzione di un comando o di un filtro. Prefisso di un operatore.

```
COMANDO -[Opzione1][Opzione2][...]

ls -al

sort -dfu $nomefile

set -- $variabile
```

```
  1 if [ $file1 -ot $file2 ]
  2 then
  3   echo "Il file $file1 è più vecchio di $file2."
  4 fi
  5
  6 if [ "$a" -eq "$b" ]
  7 then
  8   echo "$a è uguale a $b."
  9 fi
 10
 11 if [ "$c" -eq 24 -a "$d" -eq 47 ]
 12 then
 13   echo "$c è uguale a 24 e $d è uguale a 47."
 14 fi
```

-

redirezione dallo/allo `stdin` o `stdout` [trattino].

```
  1 (cd /source/directory && tar cf - . ) | (cd /dest/directory && tar
xpvf -)
  2 # Sposta l'intero contenuto di una directory in un'altra
  3 # [cortesia di Alan Cox <a.cox@swansea.ac.uk>, con una piccola
modifica]
  4
  5 # 1) cd /source/directory    Directory sorgente, dove sono contenuti
i file che
  6 #                           devono essere spostati.
  7 # 2) &&                     "lista And": se l'operazione 'cd' ha
successo,
  8 #                           allora viene eseguito il comando
successivo.
  9 # 3) tar cf - .             L'opzione 'c' del comando di
archiviazione 'tar'
 10 #                           crea un nuovo archivio, l'opzione 'f'
(file),
 11 #                           seguita da '-' designa come file di
destinazione
 12 #                           lo sdtout, e lo fa nella directory
corrente ('.').
```

```
 13 # 4) |                        Collegato a...
 14 # 5) ( ... )                 subshell
 15 # 6) cd /dest/directory      Cambia alla directory di destinazione.
 16 # 7) &&                      "lista And", come sopra
 17 # 8) tar xpvf -              Scompatta l'archivio ('x'), mantiene i
permessi e
 18 #                            le proprietà dei file ('p'), invia
messaggi
 19 #                            dettagliati allo stdout ('v'), leggendo
i dati
 20 #                            dallo stdin ('f' seguito da '-')
 21 #                            Attenzione: 'x' è un comando,
 22 #                            mentre 'p', 'v' ed 'f' sono opzioni.
 23 # Whew!
 24
 25
 26
 27 # Più elegante, ma equivalente a:
 28 #   cd source-directory
 29 #   tar cf - . | (cd ../dest/directory; tar xpvf -)
 30 #
 31 # cp -a /source/directory /dest/directory     anche questo ha lo
stesso effetto.
```

```
  1 bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
  2 #  --decomprime il file tar --   | --quindi lo passa a "tar"--
  3 #  Se "tar" non è stato aggiornato per trattare "bunzip2",
  4 #+ occorre eseguire l'operazione in due passi successivi utilizzando
una pipe.
  5 #  Lo scopo dell'esercizio è di decomprimere i sorgenti del kernel
  6 #  compressi con "bzip".
```

Va notato che, in questo contesto, il "-" non è, di per sé un operatore Bash, ma piuttosto un'opzione riconosciuta da alcune utility UNIX che scrivono allo `stdout`, come **tar**, **cat**, etc.

```
bash$ echo "qualsiasi cosa" | cat -
qualsiasi cosa
```

Dove è atteso un nome di file, `-` redirige l'output allo `stdout` (talvolta con **`tar cf`**), o accetta l'input dallo `stdin`, invece che da un file. È un metodo per utilizzare l'utility come filtro in una pipe.

```
bash$ file
Usage: file [-bciknvzL] [-f filename] [-m magicfiles] file...
```

Eseguito da solo, da riga di comando, file genera un messaggio d'errore.

Occorre aggiungere il "-" per un migliore risultato. L'esempio seguente fa sì che la shell attenda l'input dall'utente.

```
bash$ file -
abc
standard input:              ASCII text


bash$ file -
```

```
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Ora il comando accetta l'input dallo `stdin` e lo analizza.

Il "-" può essere utilizzato per collegare lo `stdout` ad altri comandi. Ciò permette alcune acrobazie, come aggiungere righe all'inizio di un file.

Utilizzare diff per confrontare un file con la *sezione* di un altro:

```
grep Linux file1 | diff file2 -
```

Infine, un esempio concreto di come usare il `-` con tar.

Esempio 3-4. Backup di tutti i file modificati il giorno precedente

```
  1 #!/bin/bash
  2
  3 #  Salvataggio di tutti i file della directory corrente che sono
stati
  4 #+ modificati nelle ultime 24 ore in un archivio "tarball" (file
trattato
  5 #+ con tar e gzip).
  6
  7 FILEBACKUP=backup-$(date +%d-%m-%Y)
  8 #                 Inserisce la data nel nome del file di
salvataggio.
  9 #                 Grazie a Joshua Tschida per l'idea.
 10 archivio=${1:-$FILEBACKUP}
 11 #  Se non viene specificato un nome di file d'archivio da riga di
comando,
 12 #+ questo verrà impostato a "backup-GG-MM-AAAA.tar.gz."
 13
 14 tar cvf - `find . -mtime -1 -type f -print` > $archivio.tar
 15 gzip $archivio.tar
 16 echo "Directory $PWD salvata nel file \"$archivio.tar.gz\"."
 17
 18
 19 #  Stephane Chazelas evidenzia che il precedente codice fallisce
l'esecuzione
 20 #+ se incontra troppi file o se un qualsiasi nome di file contiene
caratteri
 21 #+ di spaziatura.
 22
 23 # Suggerisce, quindi, le seguenti alternative:
 24 # -------------------------------------------------------------------
-
 25 #   find . -mtime -1 -type f -print0 | xargs -0 tar rvf
"$archivio.tar"
 26 #      utilizzando la versione GNU di "find".
 27
 28
 29 #   find . -mtime -1 -type f -exec tar rvf "$archivio.tar" '{}' \;
 30 #         portabile su altre versioni UNIX, ma molto più lento.
 31 # -------------------------------------------------------------------
-
 32
```

```
33
34 exit 0
```

Nomi di file che iniziano con "-" possono provocare problemi quando vengono utilizzati con il "-" come operatore di redirezione. Uno script potrebbe verificare questa possibilità ed aggiungere un prefisso adeguato a tali nomi, per esempio `./-NOMEFILE`, `$PWD/-NOMEFILE` o `$PATHNAME/-NOMEFILE`.

Anche il valore di una variabile che inizia con un `-` potrebbe creare problemi.

```
  1 var="-n"
  2 echo $var
  3 # Ha l'effetto di un  "echo -n", che non visualizza
nulla.
```

-

directory di lavoro precedente. Il comando **cd -** cambia alla directory di lavoro precedente. Viene utilizzata la variabile d'ambiente $OLDPWD.

Non bisogna confondere il "-" utilizzato in questo senso con l'operatore di redirezione "-" appena discusso. L'interpretazione del "-" dipende dal contesto in cui appare.

-

Meno. Segno meno in una operazione aritmetica.

=

Uguale. Operatore di assegnamento

```
 1 a=28
 2 echo $a   # 28
```

In un contesto differente, il simbolo di "=" è l'operatore di confronto tra stringhe.

+

Più. Addizione, operazione aritmetica.

In un contesto differente, il simbolo + è un operatore di Espressione Regolare.

+

Opzione. Opzione per un comando o un filtro.

Alcuni comandi e builtins utilizzano il segno + per abilitare certe opzioni ed il segno - per disabilitarle.

%

modulo. Modulo (resto di una divisione) , operatore aritmetico.

In un contesto differente, il simbolo % è l'operatore di ricerca di corrispondenza.

~

directory home [tilde]. Corrisponde alla variabile interna $HOME. *~bozo* è la directory home di bozo, e **ls ~bozo** elenca il suo contenuto. ~/ è la directory home dell'utente corrente e **ls ~/** elenca il suo contenuto.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~utente-inesistente
~utente-inesistente
```

~+

directory di lavoro corrente. Corrisponde alla variabile interna $PWD.

~-

directory di lavoro precedente. Corrisponde alla variabile interna $OLDPWD.

^

inizio-riga. In una espressione regolare, un "^" rinvia all'inizio di una riga di testo.

Caratteri di controllo

modificano il comportamento di un terminale o la visualizzazione di un testo. Un carattere di controllo è la combinazione di **CONTROL** + **tasto**.

Normalmente i caratteri di controllo non sono utili inseriti in uno script.

- **`Ctl-B`**

  Backspace (ritorno non distruttivo).

- **`Ctl-C`**

  Interruzione. Termina un job in primo piano.

- **`Ctl-D`**

  Uscita dalla shell (simile a exit).

"EOF" (end of file). Anch'esso termina l'input dallo `stdin`.

Durante la digitazione di un testo in una console o in una finestra xterm, **`Ctl-D`** cancella il carattere che si trova sotto al cursore. Quando non ci sono più caratteri, **`Ctl-D`** determina la prevista uscita dalla sessione.

- **`Ctl-G`**

  "SEGNALE ACUSTICO" (beep).

- **`Ctl-H`**

  Backspace (ritorno distruttivo).

```
  1 #!/bin/bash
  2 # Inserire Ctl-H in una stringa.
  3
  4 a="^H^H"                  # Due Ctl-H (backspace).
  5 echo "abcdef"             # abcdef
  6 echo -n "abcdef$a "       # abcd f
  7 # Spazio finale  ^             ^ Doppio backspace
  8 echo -n "abcdef$a"        # abcdef
  9 #  Nessuno spazio alla fine       Non viene seguito il backspace
(perché?)
 10                           # I risultati possono essere piuttosto
diversi da
 11                           #+ ciò che ci si aspetta.
 12 echo; echo
```

- **`Ctl-I`**

  Tabulazione orizzontale.

- **`Ctl-J`**

  Nuova riga (line feed).

- **`Ctl-K`**

  Tabulazione verticale.

  Durante la digitazione di un testo in una console o in una finestra xterm, **`Ctl-K`** cancella i caratteri a partire da quello che si trova sotto il cursore (compreso) fino alla fine della riga.

- **`Ctl-L`**

  Formfeed (pulisce lo schermo del terminale). Ha lo stesso effetto del comando clear.

- **`Ctl-M`**

  A capo.

```
  1 #!/bin/bash
  2 # Grazie a Lee Maschmeyer per l'esempio.
  3
  4 read -n 1 -s -p $'Control-M sposta il cursore all'inizio della riga.
Premi Invio. \x0d'
  5                          # Naturalmente, '0d' è l'equivalente
esadecimale di Control-M.
  6 echo >&2   #  '-s' non visualizza quello che viene digitato,
  7            #+ quindi è necessario andare a capo esplicitamente.
  8
  9 read -n 1 -s -p $'Control-J sposta il cursore alla riga successiva.
\x0a'
 10 echo >&2   #  Control-J indica nuova riga (linefeed).
 11
 12 ###
 13
 14 read -n 1 -s -p $'E Control-K\x0b lo sposta direttamente in basso.'
 15 echo >&2   #  Control-K indica la tabulazione verticale.
 16
 17 # Un esempio migliore dell'effetto di una tabulazione verticale è il
seguente:
 18
 19 var=$'\x0aQuesta è la riga finale\x0bQuesta è la riga iniziale\x0a'
 20 echo "$var"
 21 #  Stesso risultato dell'esempio precedente. Tuttavia:
 22 echo "$var" | col
 23 #  Questo provoca l'inversione nella visualizzazione delle righe.
 24 #  Inoltre spiega il motivo per cui sono stati posti dei line feed
all'inizio e
 25 #+ alla fine della riga: evitare una visualizzazione confusa.
 26
 27 # La spiegazione di Lee Maschmeyer:
 28 # -------------------------------
 29 #  Nel [primo esempio di tabulazione verticale] . . . questa esegue
 30 #+ una semplice visualizzazione alla riga inferiore senza il ritorno
a capo.
 31 #  Ma questo vale solo per i dispositivi, quali la console Linux,
 32 #+ che non consentono di andare "in senso inverso."
 33 #  Il vero scopo della TV è quello di andare in SÙ, non in giù.
 34 #  Ciò può essere sfruttato per stampare dei soprascritti.
 35 #  L'utility col può essere usata per simulare il corretto
comportamento
 36 #+ di una TV.
 37
 38 exit 0
```

- **Ctl-Q**

  Ripristino (XON).

  Ripristina lo `stdin` di un terminale.

- **Ctl-S**

  Sospensione (XOFF).

  Congela lo `stdin` di un terminale. (Si usi Ctl-Q per ripristinarlo.)

- **Ctl-U**

Cancella una riga di input, a partire dal cursore in senso inverso fino all'inizio della riga. In alcune impostazioni, **Ctl-U** cancella l'intera riga di input, *indipendentemente dalla posizione del cursore*.

- **Ctl-V**

  Durante la digitazione di un testo, **Ctl-V** consente l'inserimento di caratteri di controllo. Ad esempio, le due righe seguenti si equivalgono:

```
1 echo -e '\x0a'
2 echo <Ctl-V><Ctl-J>
```

  Ctl-V è particolarmnete utile in un editor di testo.

- **Ctl-W**

  Durante la digitazione di un testo in una console o in una finestra xterm, **Ctl-W** cancella a partire dal carattere che si trova sotto al cursore all'indietro fino al primo spazio incontrato. In alcune impostazioni, **Ctl-W** cancella all'indietro fino al primo carattere non alfanumerico.

- **Ctl-Z**

  Sospende un'applicazione in primo piano.

Spaziatura

serve come divisore, separando comandi o variabili. La spaziatura è formata da spazi, tabulazioni, righe vuote, o una loro qualsiasi combinazione. In alcuni contesti, quale l'assegnamento di variabile, la spaziatura non è consentita e produce un errore di sintassi.

Le righe vuote non hanno alcun affetto sull'azione dello script, sono quindi molto utili per separare visivamente le diverse sezioni funzionali.

$IFS, è la speciale variabile che separa i campi di input per determinati comandi. Il carattere preimpostato è lo spazio.

Note

[1] La shell esegue *l'espansione delle parentesi graffe*. Il comando agisce sul *risultato* dell'espansione.

[2] Eccezione: una porzione di codice tra parentesi graffe come parte di una pipe *deve* essere eseguita come subshell.

```
  1 ls | { read primariga; read secondariga; }
  2 #  Errore. Il blocco di codice tra le parentesi graffe esegue
una subshell,
  3 #+ così l'output di "ls" non può essere passato alle variabili
interne
  4 #  al blocco.
  5 echo "La prima riga è $primariga; la seconda riga è
```

```
$secondariga"
  6 # Non funziona.
  7
  8 # Grazie, S.C.
```

Introduzione alle variabili ed ai parametri

Le variabili sono il cuore di qualsiasi linguaggio di scripting e di programmazione. Compaiono nelle operazioni aritmetiche, nelle manipolazioni quantitative e nelle verifiche di stringhe e sono indispensabili per lavorare a livello di astrazione con i simboli - parole che rappresentano qualcos'altro. Una variabile non è nient'altro che una locazione, o una serie di locazioni, di memoria del computer che contiene un dato.

Sostituzione di variabile

Parte riassuntiva

Per Sfruttare il contenuto o il valore in esso memorizzato di una variabile che è chiamata sostituzione di variabile: è con questo simbolo $NOMEVAR.

Noi per assegnare un valore ad una variabile (es come se fosse un contenitore dove possimamo metterci le cose dentro e prelevarle quando servono) dobbiamo innanzitutto attribuirgli un valore, e cioè numeri o lettere ecc. Le variabili vanno anke dichiarate cioè dire all'interprete cosa c'è nella variabile stringa oppure intero, letter eo numeri interi oppure numeri con la virgola ecc.

Per esempio come faccio ad assegnare un valore ad una variabile? Ecco un esempio

```
1 #!/bin/bash
  2
  3 # Variabili: assegnamento e sostituzione
  4
  5 a=375
  6 ciao=$a
  7
  8 #-----------------------------------------------------------------------
------
  9 # Quando si inizializzano le variabili, non sono consentiti spazi
prima
 10 #+ e dopo il segno  =.
 11
 12 #  Nel caso "VARIABILE =valore",
 13 #+ lo script cerca di eseguire il comando "VARIABILE"  con
l'argomento
 14 #+ "=valore". Nel caso "VARIABILE= valore", lo script cerca di
eseguire
 15 #+ il comando "valore" con la variabile d'ambiente "VARIABILE"
impostata a "".
 16 #-----------------------------------------------------------------------
------
 17
 18
 19 echo ciao     # Non è un riferimento a variabile, ma solo la stringa
"ciao".
 20
```

```
 21 echo $ciao
 22 echo ${ciao}  # Come sopra.
 23
 24 echo "$ciao"
 25 echo "${ciao}"
 26
 27 echo
 28
 29 ciao="A B  C   D"
 30 echo $ciao    # A B C D
 31 echo "$ciao"  # A B  C   D
 32 #  Come si può vedere, echo $ciao  e  echo "$ciao" producono
 33 #+ risultati differenti. Il quoting di una variabile conserva gli
spazi.
 34
 35 echo
 36
 37 echo '$ciao'  # $ciao
 38 #  Gli apici singoli disabilitano la referenziazione alla variabile,
 39 #+ perché il simbolo "$" viene interpretato letteralmente.
 40
 41 # Notate l'effetto dei differenti tipi di quoting.
 42
 43
 44 ciao=     # Imposta la variabile al valore nullo.
 45 echo "\$ciao (valore nullo) = $ciao"
 46 #  Attenzione, impostare una variabile al valore nullo non è la
stessa
 47 #+ cosa di annullarla, sebbene il risultato finale sia lo stesso
(vedi oltre).
 48 #
 49 # ------------------------------------------------------------
 50 #
 51 #  È consentito impostare più variabili sulla stessa riga,
 52 #+ separandole con uno spazio.
 53 #  Attenzione, questa forma può diminuire la leggibilità
 54 #+ e potrebbe non essere portabile.
 55
 56 var1=variabile1  var2=variabile2  var3=variabile3
 57 echo
 58 echo "var1=$var1   var2=$var2  var3=$var3"
 59
 60 # Potrebbe causare problemi con le versioni più vecchie di "sh".
 61
 62 # ------------------------------------------------------------
 63
 64 echo; echo
 65
 66 numeri="uno due tre"
 67 altri_numeri="1 2 3"
 68 # Se ci sono degli spazi all'interno di una variabile, allora è
 69 #+ necessario il quoting.
 70 echo "numeri = $numeri"
 71 echo "altri_numeri = $altri_numeri"   # altri_numeri = 1 2 3
 72 echo
 73
 74 echo "variabile_non_inizializzata = $variabile_non_inizializzata"
 75 # Una variabile non inizializzata ha valore nullo  (nessun valore).
 76 variabile_non_inizializzata=   #  Viene dichiarata, ma non
inizializzata
 77                               #+ (è come impostarla al valore
nullo,
```

```
78                                #+ vedi sopra).
79 echo "variabile_non_inizializzata = $variabile_non_inizializzata"
80                                # Ha ancora valore nullo.
81
82 variabile_non_inizializzata=23       # È impostata.
83 unset variabile_non_inizializzata    # Viene annullata.
84 echo "variabile_non_inizializzata = $variabile_non_inizializzata"
85                                      # Ha ancora valore nullo.
86
87 echo
88
89 exit 0
```

Una variabile non inizializzata ha valore "nullo": cioè proprio nessun valore (non zero!). Utilizzare una variabile prima di averle assegnato un valore, solitamente provoca dei problemi.

Ciò nonostante è possibile eseguire operazioni aritmetiche su una variabile non inizializzata.

```
  1 echo "$non_inizializzata"                          #
(riga vuota)
  2 let "non_inizializzata += 5"                       #
Aggiunge 5 alla variabile.
  3 echo "$non_inizializzata"                          # 5
  4
  5 #  Conclusione:
  6 #  Una variabile non inizializzata non ha alcun valore,
tuttavia si comporta,
  7 #+ nelle operazioni aritmetiche, come se il suo valore
fosse 0 (zero).
  8 #  Questo è un comportamento non documentato (e
probabilmente non portabile).
```

Vedi anche Esempio 11-20.

Parte originale del manuale indispensabile

L'= E' un operatore di assegnamento se si utilizza –eq è per fare le verifiche. Da non confondere

L'= Può fare anche da operatore di verifica dipende solo da che valore gli è stato attribuito.

Assegnamento esplicito di variabile

```
 1 #!/bin/bash
 2 # Variabili nude
 3
 4 echo
 5
 6 # Quando una variabile è "nuda", cioè, senza il '$' davanti?
 7 # Durante l'assegnamento, ma non nella referenziazione.
 8
 9 # Assegnamento
10 a=879
11 echo "Il valore di \"a\" è $a."
12
13 # Assegnamento con l'utilizzo di 'let'
14 let a=16+5
```

```
15 echo "Il valore di \"a\" ora è $a."
16
17 echo
18
19 # In un ciclo 'for' (in realtà, un tipo di assegnamento mascherato)
20 echo -n "I valori di \"a\" nel ciclo sono: "
21 for a in 7 8 9 11
22 do
23   echo -n "$a "
24 done
25
26 echo
27 echo
28
29 # In un enunciato 'read' (un altro tipo di assegnamento)
30 echo -n "Immetti il valore di \"a\" "
31 read a
32 echo "Il valore di \"a\" ora è $a."
33
34 echo
35
36 exit 0
```

Esempio 4-3. Assegnamento di variabile, esplicito e indiretto

```
 1 #!/bin/bash
 2
 3 a=23              # Caso comune
 4 echo $a
 5 b=$a
 6 echo $b
 7
 8 # Ora in un modo un po' più raffinato (sostituzione di comando).
 9
10 a=`echo Ciao!`    # Assegna il risultato del comando 'echo' ad 'a'
11 echo $a
12
13 #  Nota: l'utilizzo del punto esclamativo (!) nella sostituzione di
comando
14 #+ non funziona da riga di comando, perché il (!) attiva il
15 #+ "meccanismo di cronologia" della shell Bash. All'interno di uno
script,
16 #+ però, le funzioni di cronologia sono disabilitate.
17
18 a=`ls -l`         #  Assegna il risultato del comando 'ls -l' ad 'a'
19 echo $a          #  Senza l'utilizzo del quoting vengono eliminate
20                  #+ le tabulazioni ed i ritorni a capo.
21 echo
22 echo "$a"        # L'utilizzo del quoting preserva gli spazi.
23                  # (Vedi il capitolo sul "Quoting.")
24
25 exit 0
```

Assegnamento di variabile utilizzando $(...) (metodo più recente rispetto agli apici inversi)

```
 1 # Dal file /etc/rc.d/rc.local
 2 R=$(cat /etc/redhat-release)
 3 arch=$(uname -m)
```

Le variabili Bash non sono tipizzate

La differenza dell bash tra glia ltri linguaggi di programmazione è che non c'è bisogno di digli se il contenuto di una variabile è intero o stringa.

Intero o stringa?

```
 1 #!/bin/bash
 2 # int-or-string.sh: Intero o stringa?
 3
 4 a=2334                   # Intero.
 5 let "a += 1"
 6 echo "a = $a "           # a = 2335
 7 echo                     # Intero, ancora.
 8
 9
10 b=${a/23/BB}             # Sostituisce "23" con "BB".
11                          # Questo trasforma $b in una stringa.
12 echo "b = $b"            # b = BB35
13 declare -i b             # Dichiararla come intero non aiuta.
14 echo "b = $b"            # b = BB35
15
16 let "b += 1"             # BB35 + 1 =
17 echo "b = $b"            # b = 1
18 echo
19
20 c=BB34
21 echo "c = $c"            # c = BB34
22 d=${c/BB/23}             # Sostituisce "BB" con "23".
23                          # Questo trasforma $d in un intero.
24 echo "d = $d"            # d = 2334
25 let "d += 1"             # 2334 + 1 =
26 echo "d = $d"            # d = 2335
27 echo
28
29 # Che dire a proposito delle variabili nulle?
30 e=""
31 echo "e = $e"            #  e =
32 let "e += 1"             #  Sono consentite le operazioni aritmetiche sulle
33                          #+ variabili nulle?
34 echo "e = $e"            #  e = 1
35 echo                     #  Variabile nulla trasformata in un intero.
36
37 # E sulle variabili non dichiarate?
38 echo "f = $f"            # f =
39 let "f += 1"             # Sono consentite le operazioni aritmetiche?
40 echo "f = $f"            # f = 1
41 echo                     # Variabile non dichiarata trasformata in un
intero.
42
43
44
45 # Le variabili in Bash non sono tipizzate.
46
47 exit 0
```

Questo è un vantaggio in programmazione!!!! (°J°)

Tipi speciali di variabili

variabili locali

sono variabili visibili solo all'interno di un blocco di codice o funzione (vedi anche variabili locali in funzioni)

variabili d'ambiente

sono variabili che hanno a che fare con il comportamento della shell o dell'interfaccia utente

Più in generale, ogni processo possiede un proprio "ambiente", ovvero un gruppo di variabili contenenti delle informazioni a cui il processo fa riferimento. Da questo punto di vista, la shell si comporta come qualsiasi altro processo.

Tutte le volte che la shell viene eseguita crea le variabili di shell che corrispondono alle sue variabili d'ambiente. L'aggiornamento o l'aggiunta di nuove variabili di shell provoca l'aggiornamento del suo ambiente. Tutti i processi generati dalla shell (i comandi eseguiti) ereditano questo ambiente.

Lo spazio assegnato all'ambiente è limitato. Creare troppe variabili d'ambiente, o se alcune occupano eccessivo spazio, potrebbe causare problemi.

```
bash$ eval "`seq 10000 | sed -e 's/.*/export
var&=ZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Argument list too long
```

Se uno script imposta delle variabili d'ambiente, è necessario che vengano "esportate", cioè trasferite all'ambiente dei programmi che verranno eseguiti. Questo è il compito del comando export.

Uno script può "esportare" le variabili solo verso i processi figli, vale a dire solo nei confronti dei comandi o dei processi che vengono iniziati da quel particolare script. Uno script eseguito da riga di comando *non può* esportare le variabili all'indietro, verso l'ambiente precedente. Allo stesso modo, i processi figli non possono esportare le variabili all'indietro verso i processi genitori che li hanno generati.

---

parametri posizionali

rappresentano gli argomenti passati allo script da riga di comando - $0, $1, $2, $3... $0 contiene il nome dello script, $1 è il primo argomento, $2 il secondo, $3 il terzo, ecc.. [1] Dopo $9 il numero degli argomenti deve essere racchiuso tra parentesi graffe, per esempio, ${10}, ${11}, ${12}.

Le variabili speciali $* e $@ forniscono il numero di *tutti* i parametri posizionali passati.

Esempio 4-5. Parametri posizionali

```
 1 #!/bin/bash
 2
 3 # Eseguite lo script con almeno 10 parametri, per esempio
 4 # ./nomescript 1 2 3 4 5 6 7 8 9 10
 5 MINPARAM=10
 6
 7 echo
 8
 9 echo "Il nome dello script è \"$0\"."
10 # Aggiungete ./ per indicare la directory corrente
11 echo "Il nome dello script è \"`basename $0`\"."
12 # Visualizza il percorso del nome (vedi 'basename')
13
14 echo
15
16 if [ -n "$1" ]               #  Utilizzate il quoting per la
variabile
17                              #+ da verificare.
18 then
19  echo "Il parametro #1 è $1"  # È necessario il quoting
20                              #+ per visualizzare il #
21 fi
22
23 if [ -n "$2" ]
24 then
25  echo "Il parametro #2 è $2"
26 fi
27
28 if [ -n "$3" ]
29 then
30  echo "Il parametro #3 è $3"
31 fi
32
33 # ...
34
35
36 if [ -n "${10}" ]  #  I parametri > $9 devono essere racchiusi
37                    #+ tra {parentesi graffe}.
38 then
39  echo "Il parametro #10 è ${10}"
40 fi
41
42 echo "-----------------------------------"
43 echo "In totale i parametri passati sono: "$*""
44
45 if [ $# -lt "$MINPARAM" ]
46 then
47   echo
48   echo "Lo script ha bisogno di almeno $MINPARAM argomenti da riga
di comando!"
49 fi
50
51 echo
52
53 exit 0
```

La *notazione parentesi graffe*, applicata ai parametri posizionali, può essere facilmente impiegata per la referenziazione all'*ultimo* argomento passato allo script da riga di comando. Questo richiede anche la referenziazione indiretta.

```
  1 arg=$#                      # Numero di argomenti passati.
  2 ultimo_argomento=${!args}  # Notate che ultimo_argomento=${!$#} non
funziona.
```

Alcuni script possono eseguire compiti diversi in base al nome con cui vengono invocati. Affinché questo possa avvenire, lo script ha bisogno di verificare `$0`, cioè il nome con cui è stato invocato. Naturalmente devono esserci dei link simbolici ai nomi alternativi dello script. Vedi Esempio 12-2.

Se uno script si aspetta un parametro passato da riga di comando, ma è stato invocato senza, ciò può causare un assegnamento del valore nullo alla variabile che deve essere inizializzata da quel parametro. Di solito, questo non è un risultato desiderabile. Un modo per evitare questa possibilità è aggiungere un carattere supplementare ad entrambi i lati dell'enunciato di assegnamento che utilizza il parametro posizionale.

```
  1 variabile1_=$1_
  2 #  Questo evita qualsiasi errore, anche se non è presente
  3 #+ il parametro posizionale.
  4
  5 argomento_critico01=$variabile1_
  6
  7 #  Il carattere aggiunto può essere tolto più tardi, se si
  8 #+ desidera, in questo modo:
  9 variabile1=${variabile1_/_/}   #  Si hanno effetti collaterali solo
se
 10                                #+ $variabile1_ inizia con "_".
 11 #  È stato utilizzato uno dei modelli di sostituzione di parametro
trattati
 12 #+ nel Capitolo 9. Un modo più diretto per gestire la situazione è
una
 13 #+ semplice verifica della presenza dei parametri posizionali
attesi.
 14
 15 if [ -z $1 ]
 16 then
 17   exit $MANCA_PARAM_POSIZIONALE
 18 fi
```

Esempio 4-6. verifica del nome di dominio: wh, whois

```
  1 #!/bin/bash
  2
  3 # Esegue una verifica 'whois nome-dominio' su uno dei 3 server:
  4 #                    ripe.net, cw.net, radb.net
  5
  6 # Inserite questo script, con nome 'wh', nel file /usr/local/bin
  7
  8 # Sono richiesti i seguenti link simbolici:
  9 # ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
 10 # ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
 11 # ln -s /usr/local/bin/wh /usr/local/bin/wh-radb
 12
 13
 14 if [ -z "$1" ]
 15 then
```

```
 16   echo "Utilizzo: `basename $0` [nome-dominio]"
 17   exit 65
 18 fi
 19
 20 case `basename $0` in
 21 # Verifica il nome dello script e interroga il server adeguato
 22     "wh"     ) whois $1@whois.ripe.net;;
 23     "wh-ripe") whois $1@whois.ripe.net;;
 24     "wh-radb") whois $1@whois.radb.net;;
 25     "wh-cw"  ) whois $1@whois.cw.net;;
 26     *        ) echo "Utilizzo: `basename $0` [nome-dominio]";;
 27 esac
 28
 29 exit 0
```

Il comando **shift** riassegna i parametri posizionali, spostandoli di una posizione verso sinistra.

`$1` <--- `$2`, `$2` <--- `$3`, `$3` <--- `$4`, etc.

Il vecchio `$1` viene sostituito, ma *`$0` (il nome dello script) non cambia*. Se si utilizza un numero elevato di parametri posizionali, **shift** permette di accedere a quelli dopo il `9`, sebbene questo sia possibile anche con la notazione {parentesi graffe}.

Esempio 4-7. Utilizzare shift

```
  1 #!/bin/bash
  2 # Utilizzo di 'shift' per elaborare tutti i parametri posizionali.
  3
  4 #  Chiamate lo script shft ed invocatelo con alcuni parametri, per
esempio
  5 #          ./shft a b c def 23 skidoo
  6
  7 until [ -z "$1" ]  # Finché ci sono parametri...
  8 do
  9   echo -n "$1 "
 10   shift
 11 done
 12
 13 echo               # Linea extra.
 14
 15 exit 0
```

Il comando **shift** funziona anche sui parametri passati ad una funzione. Vedi Esempio 34-12.

Note

[1] È il processo che chiama lo script che imposta il parametro *`$0`*. Per convenzione, questo parametro è il nome dello script. Vedi la pagina di manuale di **execv**.

Quoting

Con il termine "quoting" si intende semplicemente questo: inserire una stringa tra apici. Viene utilizzato per proteggere i caratteri speciali contenuti nella stringa dalla reinterpretazione o espansione da parte della shell o di uno script. (Un carattere si definisce "speciale" se viene interpretato diversamente dal suo significato letterale, come il carattere jolly *.)

```
bash$ ls -l [Vv]*
 -rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

Tuttavia alcuni programmi ed utility possono ancora reinterpretare o espandere i caratteri speciali contenuti in una stringa a cui è stato applicato il quoting. Un utilizzo importante del quoting è quello di proteggere un parametro passato da riga di comando dalla reinterpretazione da parte della shell, ma permettere ancora al programma chiamante di espanderlo.

```
bash$ grep '[Pp]rima' *.txt
file1.txt:Questa è la prima riga di file1.txt.
file2.txt:Questa è la prima riga di file2.txt.
```

È da notare che l'istruzione **`grep [Pp]rima *.txt`**, senza quoting, funziona con la shell Bash, ma *non* con **tcsh**.

Nella referenziazione di una variabile, è generalmente consigliabile racchiudere la variabile referenziata tra apici doppi (" "). Questo preserva dall'interpretazione tutti i caratteri speciali della variabile, tranne $, ` (apice inverso), e \ (escape). [1] Mantenere il $ come carattere speciale consente la referenziazione di una variabile racchiusa tra doppi apici (*"$variabile"*), cioè, sostituire la variabile con il suo valore (vedi Esempio 4-1, precedente).

L'utilizzo degli apici doppi previene la suddivisione delle parole. [2] Un argomento tra apici doppi viene considerato come un'unica parola, anche se contiene degli spazi.

```
 1 variabile1="una variabile contenente cinque parole"
 2 COMANDO Questa è $variabile1    # Esegue COMANDO con 7 argomenti:
 3 # "Questa" "è" "una" "variabile" "contenente" "cinque" "parole"
 4
 5 COMANDO "Questa è $variabile1"  # Esegue COMANDO con 1 argomento:
 6 # "Questa è una variabile contenente cinque parole"
 7
 8
 9 variabile2=""    # Vuota.
10
11 COMANDO $variabile2 $variabile2 $variabile2
12 # Esegue COMANDO con nessun argomento.
13
14 COMANDO "$variabile2" "$variabile2" "$variabile2"
15 # Esegue COMANDO con 3 argomenti vuoti.
16
17 COMANDO "$variabile2 $variabile2 $variabile2"
18 # Esegue COMANDO con 1 argomento (2 spazi).
19
20 # Grazie, S.C.
```

È necessario porre gli argomenti tra doppi apici in un enunciato **echo** solo quando la

suddivisione delle parole può rappresentare un problema.

Esempio 5-1. Visualizzare variabili strane

```
 1 #!/bin/bash
 2 # weirdvars.sh: Visualizzare strane variabili.
 3
 4 var="'(]\\{}\$\""
 5 echo $var        # '(]\{}$"
 6 echo "$var"      # '(]\{}$"     Nessuna differenza.
 7
 8 echo
 9
10 IFS='\'
11 echo $var        # '(] {}$"     \ trasformata in spazio.
12 echo "$var"      # '(]\{}$"
13
14 # Esempi forniti da S.C.
15
16 exit 0
```

Gli apici singoli (' ') agiscono in modo simile a quelli doppi, ma non consentono la referenziazione alle variabili, perché non è più consentita la reinterpretazione di $. All'interno degli apici singoli, *tutti* i caratteri speciali, tranne ', vengono interpretati letteralmente. Gli apici singoli ("quoting pieno") rappresentano un metodo di quoting più restrittivo di quello con apici doppi ("quoting parziale").

Dal momento che anche il carattere di escape (\) viene interpretato letteralmente, effettuare il quoting di apici singoli mediante apici singoli non produce il risultato atteso.

```
 1 echo "Why can't I write 's between single quotes"
 2 # Perché non riesco a scrivere 's tra apici singoli
 3
 4 echo
 5
 6 # Metodo indiretto.
 7 echo 'Why can'\''t I write '"'"'s between single quotes'
 8 #    |-------|  |----------|   |----------------------|
 9 #  Tre stringhe tra apici singoli, con il carattere di escape e
10 #+ apice singolo in mezzo.
11
12 #  Esempio cortesemente fornito da Stephane Chazelas.
```

L'escaping è un metodo per effettuare il quoting di un singolo carattere. Il carattere di escape (\), posto davanti ad un altro carattere, informa la shell che quest'ultimo deve essere interpretato letteralmente.

Con alcuni comandi e utility, come echo e sed, l'escaping di un carattere potrebbe avere un effetto particolare - quello di attribuire un significato specifico a quel carattere (le c.d. sequenze di escape [N.d.T.]).

Significati speciali di alcuni caratteri preceduti da quello di escape:

Da usare con **echo** e **sed**

\n

significa a capo

\r

significa invio

\t

significa tabulazione

\v

significa tabulazione verticale

\b

significa ritorno (backspace)

\a

"significa allerta" (segnale acustico o accensione di un led)

\0xx

trasforma in carattere ASCII il valore ottale *0xx*

Esempio 5-2. Sequenze di escape

```
  1 #!/bin/bash
  2 # escaped.sh: sequenze di escape
  3
  4 echo; echo
  5
  6 echo "\v\v\v\v\v"      # Visualizza letteralmente: \v\v\v\v\v .
  7 #  Utilizzate l'opzione  -e  con 'echo' per un corretto impiego
delle
  8 #+ sequenze di escape.
  9 echo "=============="
 10 echo "TABULAZIONE VERTICALE"
 11 echo -e "\v\v\v\v\v"   # Esegue 4 tabulazioni verticali.
 12 echo "=============="
 13
 14 echo "VIRGOLETTE"
 15 echo -e "\042"       #  Visualizza " (42 è il valore ottale del
 16                      #+ carattere ASCII virgolette).
 17 echo "=============="
 18
 19 # Il costrutto $'\X'  rende l'opzione -e superflua.
 20 echo; echo "A_CAPO E BEEP"
 21 echo $'\n'           # A capo.
 22 echo $'\a'           # Allerta (beep).
 23
 24 echo "==============="
```

```
 25 echo "VIRGOLETTE"
 26 # La versione 2 e successive di Bash consente l'utilizzo del
costrutto $'\nnn'.
 27 # Notate che in questo caso, '\nnn' è un valore ottale.
 28 echo $'\t \042 \t'   # Virgolette (") tra due tabulazioni.
 29
 30 # Può essere utilizzato anche con valori esadecimali nella forma
$'\xhhh'.
 31 echo $'\t \x22 \t'   # Virgolette (") tra due tabulazioni.
 32 # Grazie a Greg Keraunen per la precisazione.
 33 # Versioni precedenti di Bash consentivano '\x022'.
 34 echo "==============="
 35 echo
 36
 37
 38 # Assegnare caratteri ASCII ad una variabile.
 39 # ----------------------------------------
 40 virgolette=$'\042'   # " assegnate alla variabile.
 41 echo "$virgolette Questa è una stringa tra virgolette $virgolette, \
 42 mentre questa parte è al di fuori delle virgolette."
 43
 44 echo
 45
 46 # Concatenare caratteri ASCII in una variabile.
 47
 48 tripla_sottolineatura=$'\137\137\137'
 49 # 137 è il valore ottale del carattere ASCII  '_'.
 50 echo "$tripla_sottolineatura SOTTOLINEA  $tripla_sottolineatura"
 51
 52 echo
 53
 54 ABC=$'\101\102\103\010'
 55 # 101, 102, 103 sono i valori ottali di A, B, C.
 56
 57 echo $ABC
 58
 59 echo; echo
 60
 61 escape=$'\033'
 62 # 033 è il valore ottale del carattere di escape.
 63
 64 echo "\"escape\" visualizzato come $escape"
 65 # nessun output visibile.
 66
 67 echo; echo
 68
 69 exit 0
```

Vedi Esempio 35-1 per un'altra dimostrazione di **$' '** come costrutto di espansione di stringa.

\"

mantiene il significato letterale dei doppi apici

```
  1 echo "Ciao"                 # Ciao
  2 echo "\"Ciao\", disse."     # "Ciao", disse.
```

\$

mantiene il significato letterale del segno del dollaro (la variabile che segue \$ non verrà referenziata)

```
  1 echo "\$variabile01"  # visualizza $variabile01
```

\\

mantiene il significato letterale della barra inversa

```
  1 echo "\\"  # visualizza \
  2
  3 # Mentre . . .
  4
  5 echo "\"   # Invoca il prompt secondario da riga di
comando.
  6           # In uno script provoca un messaggio d'errore.
```

Il comportamento della \ dipende dal contesto: se le è stato applicato l'escaping o il quoting, se appare all'interno di una sostituzione di comando o in un here document.

```
  1                       #  Escaping e quoting semplice
  2 echo \z               #  z
  3 echo \\z              # \z
  4 echo '\z'             # \z
  5 echo '\\z'            # \\z
  6 echo "\z"             # \z
  7 echo "\\z"            # \z
  8
  9                       #  Sostituzione di comando
 10 echo `echo \z`        #  z
 11 echo `echo \\z`       #  z
 12 echo `echo \\\z`      # \z
 13 echo `echo \\\\z`     # \z
 14 echo `echo \\\\\\z`   # \z
 15 echo `echo \\\\\\\z`  # \\z
 16 echo `echo "\z"`      # \z
 17 echo `echo "\\z"`     # \z
 18
 19                       # Here document
 20 cat <<EOF
 21 \z
 22 EOF                   # \z
 23
 24 cat <<EOF
 25 \\z
 26 EOF                   # \z
 27
 28 # Esempi forniti da Stephane Chazelas.
```

L'escaping può essere applicato anche ai caratteri di una stringa assegnata ad una variabile, ma non si può assegnare ad una variabile il solo carattere di escape.

```
  1 variabile=\
  2 echo "$variabile"
  3 # Non funziona - dà un messaggio d'errore:
  4 # test.sh: : command not found
  5 # Un escape "nudo" non può essere assegnato in modo sicuro ad
una variabile.
  6 #
```

```
  7 #  Quello che avviene effettivamente qui è che la "\" esegue
l'escape
  8 #+ del a capo e l'effetto è         variabile=echo "$variabile"
  9 #+                                  assegnamento di variabile
non valido
 10
 11 variabile=\
 12 23skidoo
 13 echo "$variabile"       #  23skidoo
 14                         #  Funziona, perché nella seconda riga
 15                         #+ è presente un assegnamento di
variabile valido.
 16
 17 variabile=\
 18 #         \^    escape seguito da uno spazio
 19 echo "$variabile"       # spazio
 20
 21 variabile=\\
 22 echo "$variabile"       # \
 23
 24 variabile=\\\
 25 echo "$variabile"
 26 # Non funziona - dà un messaggio d'errore:
 27 # test.sh: \: command not found
 28 #
 29 #  Il primo carattere di escape esegue l'escaping del secondo,
mentre il terzo
 30 #+ viene lasciato "nudo", con l'identico risultato del primo
esempio visto
 31 #+ sopra.
 32
 33 variabile=\\\\
 34 echo "$variabile"       #  \\
 35                         #  Il secondo ed il quarto sono stati
preservati dal
 36                         #+ primo e dal terzo.
 37                         #  Questo va bene.
```

L'escaping dello spazio evita la suddivisione delle parole di un argomento contenente un elenco di comandi.

```
  1 elenco_file="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
  2 # Elenco di file come argomento(i) di un comando.
  3
  4 # Aggiunge due file all'elenco, quindi visualizza tutto.
  5 ls -l /usr/X11R6/bin/xsetroot /sbin/dump $elenco_file
  6
  7 echo "------------------------------------------------------------------------
---"
  8
  9 # Cosa succede se si effettua l'escaping dei due spazi?
 10 ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $elenco_file
 11 #  Errore: i primi tre file vengono concatenati e considerati come un unico
 12 #+ argomento per 'ls -l' perché l'escaping dei due spazi impedisce la
 13 #+ divisione degli argomenti (parole).
```

Il carattere di escape rappresenta anche un mezzo per scrivere comandi su più righe. Di solito, ogni riga rappresenta un comando differente, ma il carattere di escape posto in fine di riga *effettua*

l'escaping del carattere a capo, in questo modo la sequenza dei comandi continua alla riga successiva.

```
 1 ((cd /source/directory && tar cf - . ) | \
 2 (cd /dest/directory && tar xpvf -)
 3 # Ripetizione del comando copia di un albero di directory di Alan Cox,
 4 # ma suddiviso su due righe per aumentarne la leggibilità.
 5
 6 # Come alternativa:
 7 tar cf - -C /source/directory . |
 8 tar xpvf - -C /dest/directory
 9 # Vedi la nota più sotto.
10 #(Grazie, Stephane Chazelas.)
```

Se una riga dello script termina con | (pipe) allora la \ (l' escape), non è obbligatorio. È, tuttavia, buona pratica di programmazione utilizzare sempre l'escape alla fine di una riga di codice che continua nella riga successiva.

```
 1 echo "foo
 2 bar"
 3 #foo
 4 #bar
 5
 6 echo
 7
 8 echo 'foo
 9 bar'    # Ancora nessuna differenza.
10 #foo
11 #bar
12
13 echo
14
15 echo foo\
16 bar     # Eseguito l'escaping del carattere a capo.
17 #foobar
18
19 echo
20
21 echo "foo\
22 bar"     #  Stesso risultato, perché \ viene ancora interpretato come escape
23           #+ quando è posto tra apici doppi.
24 #foobar
25
26 echo
27
28 echo 'foo\
29 bar'     #  Il carattere di escape \ viene interpretato letteralmente a
causa
30           #+ del quoting forte.
31 #foo\
32 #bar
33
34 # Esempi suggeriti da Stephane Chazelas.
```

Note

[1] Racchiudere il "!" tra doppi apici provoca un errore se usato *da riga di comando*. Viene interpretato come un comando di cronologia. In uno script, tuttavia, questo problema non si presenta, dal momento che la cronologia dei comandi di Bash è didabilitata.

Più interessante è il comportamento incoerente della "\" quando si trova tra i doppi apici.

```
bash$ echo ciao\!
ciao!


bash$ echo "ciao\!"
ciao\!




bash$ echo -e x\ty
xty


bash$ echo -e "x\ty"
x       y
```

(Grazie a Wayne Pollock per la precisazione.)

[2] La "divisione delle parole", in questo contesto, significa suddividere una stringa di caratteri in un certo numero di argomenti separati e distinti

Exit ed exit status

...there are dark corners in the Bourne shell, and people use all of them.

Chet Ramey

Il comando **exit** può essere usato per terminare uno script, proprio come in un programma in linguaggio C. Può anche restituire un valore disponibile al processo genitore dello script.

Ogni comando restituisce un *exit status* (talvolta chiamato anche *return status*). Un comando che ha avuto successo restituisce 0, mentre, in caso di insuccesso, viene restituito un valore diverso da zero, che solitamente può essere interpretato come un codice d'errore. Comandi, programmi e utility UNIX correttamente eseguiti restituiscono come codice di uscita 0, con significato di successo, sebbene ci possano essere delle eccezioni.

In maniera analoga, sia le funzioni all'interno di uno script che lo script stesso, restituiscono un exit status che nient'altro è se non l'exit status dell'ultimo comando eseguito dalla funzione o dallo script. In uno script, il comando **`exit` *nnn*** può essere utilizzato per inviare l'exit status *nnn* alla shell (*nnn* deve essere un numero decimale compreso nell'intervallo 0 - 255).

Quando uno script termina con **exit** senza alcun parametro, l'exit status dello script è quello dell'ultimo comando eseguito (*non* contando l'**exit**).

```
1 #!/bin/bash
2
3 COMANDO_1
4
5 . . .
```

```
 6
 7 # Esce con lo status dell'ultimo comando.
 8 ULTIMO_COMANDO
 9
10 exit
```

L'equivalente del solo **exit** è **exit $?** o, addirittura, tralasciando semplicemente **exit**.

```
 1 #!/bin/bash
 2
 3 COMANDO_1
 4
 5 . . .
 6
 7 # Esce con lo status dell'ultimo comando.
 8 ULTIMO_COMANDO
 9
10 exit $?
```

```
 1 #!/bin/bash
 2
 3 COMANDO1
 4
 5 . . .
 6
 7 # Esce con lo status dell'ultimo comando.
 8 ULTIMO_COMANDO
```

`$?` legge l'exit status dell'ultimo comando eseguito. Dopo l'esecuzione di una funzione, `$?` fornisce l'exit status dell'ultimo comando eseguito nella funzione. Questo è il modo che Bash ha per consentire alle funzioni di restituire un "valore di ritorno". Al termine di uno script, digitando `$?` da riga di comando, si ottiene l'exit status dello script, cioè, dell'ultimo comando eseguito che, per convenzione, è **`0`** in caso di successo o un intero tra 1 e 255 in caso di errore.

Esempio 6-1. exit / exit status

```
 1 #!/bin/bash
 2
 3 echo ciao
 4 echo $?    # Exit status 0 perché il comando è stato eseguito con successo.
 5
 6 lskdf      #  Comando sconosciuto.
 7 echo $?    #  Exit status diverso da zero perché il comando non ha
 8            #+ avuto successo.
 9
10 echo
11
12 exit 113   # Restituirà 113 alla shell.
13            # Per verificarlo, digitate "echo $?" dopo l'esecuzione dello
script.
14
15 #  Convenzionalmente, un 'exit 0' indica successo,
16 #+ mentre un valore diverso significa errore o condizione anomala.
```

$? è particolarmente utile per la verifica del risultato di un comando in uno script (vedi Esempio 12-30 e Esempio 12-16).

Il !, l'operatore logico "not", inverte il risultato di una verifica o di un comando e questo si riperquote sul relativo exit status.

Esempio 6-2. Negare una condizione utilizzando !

```
 1 true  # il builtin "true".
 2 echo "exit status di \"true\" = $?"     # 0
 3
 4 ! true
 5 echo "exit status di \"! true\" = $?"   # 1
 6 # Notate che "!" deve essere seguito da uno spazio.
 7 # !true restituisce l'errore  "command not found"
 8 #
 9 #  L'operatore '!' anteposto ad un comando richiama la cronokogia dei
10 #+ comandi di Bash.
11
12 true
13 !true
14 # Questa volta nessun errore, ma neanche nessuna negazione.
15 # Viene ripetuto semplicemente il comando precedente (true).
16
17 # Grazie a Stephane Chazelas e Kristopher Newsome.
```

Alcuni codici di exit status hanno significati riservati e non dovrebbero quindi essere usati dall'utente in uno script.

Verifiche

Qualsiasi linguaggio di programmazione, che a ragione possa definirsi completo, deve consentire la verifica di una condizione e quindi comportarsi in base al suo risultato. Bash possiede il comando **test**, vari operatori parentesi quadre, parentesi rotonde e il costrutto **if/then**.

7.1. Costrutti condizionali

- Il costrutto **if/then** verifica se l'exit status di un elenco di comandi è 0 (perché 0 significa "successo" per convenzione UNIX) e se questo è il caso, esegue uno o più comandi.
- Esiste il comando specifico **[** (parentesi quadra aperta). È sinonimo di **test** ed è stato progettato come builtin per ragioni di efficienza. Questo comando considera i suoi argomenti come espressioni di confronto, o di verifica di file, e restituisce un exit status corrispondente al risultato del confronto (0 per vero, 1 per falso).
- Con la versione 2.02, Bash ha introdotto [[...]], *comando di verifica estesa*, che esegue confronti in un modo più familiare ai programmatori in altri linguaggi. Notate che **[[** è una parola chiave, non un comando.

  Bash vede **`[[ $a -lt $b ]]`** come un unico elemento che restituisce un exit status.

  Anche i costrutti ((...)) e let ... restituiscono exit status 0 se le espressioni aritmetiche valutate sono espanse ad un valore diverso da zero. Questi costrutti di espansione aritmetica possono, quindi, essere usati per effettuare confronti aritmetici.

```
1 let "1<2" restituisce 0 (poiché "1<2" espande a "1")
2 (( 0 && 1 )) restituisce 1 (poiché "0 && 1" espande a "0")
```

- Un costrutto **if** può verificare qualsiasi comando, non solamente le condizioni comprese tra parentesi quadre.

```
 1 if cmp a b &> /dev/null  # Sopprime l'output.
 2 then echo "I file a e b sono identici."
 3 else echo "I file a e b sono diversi."
 4 fi
 5
 6 # L'utilissimo costrutto "if-grep":
 7 # ------------------------------
 8 if grep -q Bash file
 9 then echo "Il file contiene almeno un'occorrenza di Bash."
10 fi
11
12 parola=Linux
13 sequenza_lettere=inu
14 if echo "$parola" | grep -q "$sequenza_lettere"
15 # L'opzione "-q" di grep elimina l'output.
16 then
17   echo "$sequenza_lettere trovata in $parola"
18 else
19   echo "$sequenza_lettere non trovata in $parola"
20 fi
21
22
23 if COMANDO_CON_EXIT_STATUS_0_SE_NON_SI_VERIFICA_UN_ERRORE
24 then echo "Comando eseguito."
25 else echo "Comando fallito."
26 fi
```

- Un costrutto **if/then** può contenere confronti e verifiche annidate.

```
 1 if echo "Il prossimo *if* è parte del costrutto del primo *if*."
 2
 3   if [[ $confronto = "intero" ]]
 4     then (( a < b ))
 5   else
 6     [[ $a < $b ]]
 7   fi
 8
 9 then
10   echo '$a è inferiore a $b'
11 fi
```

- *Dettagliata spiegazione della "condizione-if" cortesia di Stephane Chazelas.*

Esempio 7-1. Cos'è vero?

```
 1 #!/bin/bash
 2
 3 echo
 4
 5 echo "Verifica \"0\""
 6 if [ 0 ]      # zero
 7 then
 8   echo "0 è vero."
 9 else
10   echo "0 è falso."
```

```
fi            # 0 è vero.

echo

echo "Verifica \"1\""
if [ 1 ]      # uno
then
  echo "1 è vero."
else
  echo "1 è falso."
fi            # 1 è vero.

echo

echo "Verifica \"-1\""
if [ -1 ]     # meno uno
then
  echo "-1 è vero."
else
  echo "-1 è falso."
fi            # -1 è vero.

echo

echo "Verifica \"NULL\""
if [ ]        # NULL (condizione vuota)
then
  echo "NULL è vero."
else
  echo "NULL è falso."
fi            # NULL è falso.

echo

echo "Verifica \"xyz\""
if [ xyz ]    # stringa
then
  echo "La stringa casuale è vero."
else
  echo "La stringa casuale è falso."
fi            # La stringa casuale è vero.

echo

echo "Verifica \"\$xyz\""
if [ $xyz ]   # Verifica se $xyz è nulla, ma...
              # è solo una variabile non inizializzata.
then
  echo "La variabile non inizializzata è vero."
else
  echo "La variabile non inizializzata è falso."
fi            # La variabile non inizializzata è falso.

echo

echo "Verifica \"-n \$xyz\""
if [ -n "$xyz" ]            # Più corretto, ma pedante.
then
  echo "La variabile non inizializzata è vero."
else
  echo "La variabile non inizializzata è falso."
fi            # La variabile non inizializzata è falso.
```

```
 73
 74 echo
 75
 76
 77 xyz=          # Inizializzata, ma impostata a valore nullo.
 78
 79 echo "Verifica \"-n \$xyz\""
 80 if [ -n "$xyz" ]
 81 then
 82   echo "La variabile nulla è vero."
 83 else
 84   echo "La variabile nulla è falso."
 85 fi            # La variabile nulla è falso.
 86
 87
 88 echo
 89
 90
 91 # Quando "falso" è vero?
 92
 93 echo "Verifica \"falso\""
 94 if [ "falso" ]              #  Sembra che "falso" sia solo una stringa.
 95 then
 96   echo "\"falso\" è vero."  # e verifica se è vero.
 97 else
 98   echo "\"falso\" è falso."
 99 fi            # "falso" è vero.
100
101 echo
102
103 echo "Verifica \"\$falso\""  # Ancora variabile non inizializzata.
104 if [ "$falso" ]
105 then
106   echo "\"\$falso\" è vero."
107 else
108   echo "\"\$falso\" è falso."
109 fi            # "$falso" è falso.
110               # Ora abbiamo ottenuto il risultato atteso.
111
112
113 echo
114
115 exit 0
```

Esercizio. Spiegare il comportamento del precedente Esempio 7-1.

```
 1 if [ condizione-vera ]
 2 then
 3    comando 1
 4    comando 2
 5    ...
 6 else
 7    #  Opzionale (può anche essere omesso).
 8    #  Aggiunge un determinato blocco di codice che verrà eseguito se la
 9    #+ condizione di verifica è falsa.
10    comando 3
11    comando 4
12    ...
13 fi
```


Quando *if* e *then* sono sulla stessa riga, occorre mettere un punto e virgola dopo l'enunciato *if*

per indicarne il termine. Sia *if* che *then* sono parole chiave. Le parole chiave (o i comandi) iniziano gli enunciati e prima che un nuovo enunciato possa incominciare sulla stessa riga, è necessario che il precedente venga terminato.

```
1 if [ -x "$nome_file" ]; then
```

Else if ed elif

elif

`elif` è la contrazione di else if. Lo scopo è quello di annidare un costrutto if/then in un altro.

```
 1 if [ condizione1 ]
 2 then
 3    comando1
 4    comando2
 5    comando3
 6 elif [ condizione2 ]
 7 # Uguale a else if
 8 then
 9    comando4
10    comando5
11 else
12    comando-predefinito
13 fi
```

Il costrutto **`if test condizione-vera`** è l'esatto equivalente di **`if [ condizione-vera ]`**. In quest'ultimo costrutto, la parentesi quadra sinistra **[**, è un simbolo che invoca il comando **test**. La parentesi quadra destra di chiusura, **]**, non dovrebbe essere necessaria. Ciò nonostante, le più recenti versioni di Bash la richiedono.


Il comando **test** è un builtin Bash che verifica i tipi di file e confronta le stringhe. Di conseguenza, in uno script Bash, **test** *non* richiama l'eseguibile esterno `/usr/bin/test`, che fa parte del pacchetto *sh-utils*. In modo analogo, [non chiama `/usr/bin/[`, che è un link a `/usr/bin/test`.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

Esempio 7-2. Equivalenza di test, `/usr/bin/test`, [ ] e `/usr/bin/[`

```
1 #!/bin/bash
2
3 echo
```

```
 4
 5 if test -z "$1"
 6 then
 7   echo "Nessun argomento da riga di comando."
 8 else
 9   echo "Il primo argomento da riga di comando è $1."
10 fi
11
12 echo
13
14 if /usr/bin/test -z "$1"      # Stesso risultato del builtin "test".
15 then
16   echo "Nessun argomento da riga di comando."
17 else
18   echo "Il primo argomento da riga di comando è $1."
19 fi
20
21 echo
22
23 if [ -z "$1" ]               #  Funzionalità identica al precedente blocco
24                              #+ di codice.
25 #   if [ -z "$1"                dovrebbe funzionare, ma...
26 #+  Bash risponde con il messaggio d'errore di missing close-bracket.
27 then
28   echo "Nessun argomento da riga di comando."
29 else
30   echo "Il primo argomento da riga di comando è $1."
31 fi
32
33 echo
34
35 if /usr/bin/[ -z "$1"         # Ancora, funzionalità identica alla
precedente.
36 # if /usr/bin/[ -z "$1" ]     # Funziona, ma dà un messaggio d'errore.
37 then
38   echo "Nessun argomento da riga di comando."
39 else
40   echo "Il primo argomento da riga di comando è $1."
41 fi
42
43 echo
44
45 exit 0
```

Il costrutto [[ ]] è la versione Bash più versatile di [ ]. È il *comando di verifica esteso*, adottato da *ksh88*.

Non può aver luogo alcuna espansione di nome di file o divisione di parole tra [[e]], mentre sono consentite l'espansione di parametro e la sostituzione di comando.

```
1 file=/etc/passwd
2
3 if [[ -e $file ]]
4 then
5   echo "Il file password esiste."
6 fi
```

L'utilizzo del costrutto di verifica **[[...]]** al posto di **[...]** può evitare molti errori logici negli script. Per esempio, gli operatori &&, ||, < e > funzionano correttamente in una verifica [[ ]], mentre potrebbero dare degli errori con il costrutto [ ] .

Dopo un **if** non sono strettamente necessari né il comando **test** né i costrutti parentesi quadre ([ ] o [[ ]]).

```
  1 dir=/home/bozo
  2
  3 if cd "$dir" 2>/dev/null; then # "2>/dev/null" sopprime il messaggio
d'errore.
  4   echo "Ora sei in $dir."
  5 else
  6   echo "Non riesco a cambiare in $dir."
  7 fi
```

Il costrutto "if COMANDO" restituisce l'exit status di COMANDO.

Per questo motivo, una condizione tra parentesi quadre può essere utilizzata da sola, senza **if**, se abbinata ad un costrutto lista.

```
  1 var1=20
  2 var2=22
  3 [ "$var1" -ne "$var2" ] && echo "$var1 è diversa da $var2"
  4
  5 home=/home/bozo
  6 [ -d "$home" ] || echo "La directory $home non esiste."
```

Il costrutto (( )) espande e valuta un'espressione aritmetica. Se il risultato della valutazione dell'espressione è zero, viene restituito come exit status 1, ovvero "falso". Una valutazione diversa da zero restituisce come exit status 0, ovvero "vero". Questo è in contrasto marcato con l'utilizzo di **test** e dei costrutti [ ] precedentemente discussi.

Esempio 7-3. Verifiche aritmetiche utilizzando (( ))

```
  1 #!/bin/bash
  2 # Verifiche aritmetiche.
  3
  4 # Il costrutto (( ... )) valuta e verifica le espressioni aritmetiche.
  5 # Exit status opposto a quello fornito dal costrutto  [ ... ]!
  6
  7 (( 0 ))
  8 echo "l'exit status di \"(( 0 ))\" è $?."     # 1
  9
 10 (( 1 ))
 11 echo "L'exit status di \"(( 1 ))\" è $?."     # 0
 12
 13 (( 5 > 4 ))                                   # vero
 14 echo "L'exit status di \"(( 5 > 4 ))\" è $?." # 0
 15
 16 (( 5 > 9 ))                                   # falso
 17 echo "L'exit status di \"(( 5 > 9 ))\" è $?." # 1
 18
 19 (( 5 - 5 ))                                   # 0
 20 echo "L'exit status di \"(( 5 - 5 ))\" è $?." # 1
 21
 22 (( 5 / 4 ))                                   # Divisione o.k.
 23 echo "L'exit status di \"(( 5 / 4 ))\" è $?." # 0
 24
 25 (( 1 / 2 ))                                   # Risultato della divisione
<1.
```

```
 26 echo "L'exit status di \"(( 1 / 2 ))\" è $?." # Arrotondato a 0.
 27                                               # 1
 28
 29 (( 1 / 0 )) 2>/dev/null                       # Divisione per 0 non
consentita.
 30 echo "L'exit status di \"(( 1 / 0 ))\" è $?." # 1
 31
 32 # Che funzione ha "2>/dev/null"?
 33 # Cosa succederebbe se fosse tolto?
 34 # Toglietelo, quindi rieseguite lo script.
 35
 36 exit 0
```

Operatori di verifica di file

Restituiscono vero se...

-e

il file esiste

-f

il file è un file *regolare* (non una directory o un file di dispositivo)

-s

il file ha dimensione superiore a zero

-d

il file è una directory

-b

il file è un dispositivo a blocchi (floppy, cdrom, ecc.)

-c

il file è un dispositivo a caratteri (tastiera, modem, scheda audio, ecc.)

-p

il file è una pipe

-h

il file è un link simbolico

-L

il file è un link simbolico

-S

il file è un socket

-t

il file (descrittore) è associato ad un terminale

Questa opzione può essere utilizzata per verificare se lo `stdin` (**`[ -t 0 ]`**) o lo `stdout` (**`[ -t 1 ]`**) in un dato script è un terminale.

-r

il file ha il permesso di lettura (*per l'utente che esegue la verifica*)

-w

il file ha il permesso di scrittura (per l'utente che esegue la verifica)

-x

il file ha il permesso di esecuzione (per l'utente che esegue la verifica)

-g

è impostato il bit set-group-id (sgid) su un file o directory

Se una directory ha il bit *`sgid`* impostato, allora un file creato in quella directory appartiene al gruppo proprietario della directory, non necessariamente al gruppo dell'utente che ha creato il file. Può essere utile per una directory condivisa da un gruppo di lavoro.

-u

è impostato il bit set-user-id (suid) su un file

Un file binario di proprietà di *root* con il bit *`set-user-id`* impostato funziona con i privilegi di *root* anche quando è invocato da un utente comune. [1] È utile con eseguibili (come **pppd** e **cdrecord**) che devono accedere all'hardware del sistema. Non impostando il bit *suid*, questi eseguibili non potrebbero essere invocati da un utente diverso da root.

```
              -rwsr-xr-t    1 root       178236 Oct  2  2000
/usr/sbin/pppd
```

Un file con il bit *`suid`* impostato è visualizzato con una *s* nell'elenco dei permessi.

-k

è impostato lo *`sticky bit`*

Comunemente conosciuto come "sticky bit", il bit *save-text-mode* è un tipo particolare di permesso. Se un file ha il suddetto bit impostato, quel file verrà mantenuto nella memoria cache, per consentirne un accesso più rapido. [2] Se impostato su una directory ne limita il permesso di scrittura. Impostando lo sticky bit viene aggiunta una *t* all'elenco dei permessi di un file o di una directory.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

Se l'utente non è il proprietario della directory con lo sticky bit impostato, ma ha il permesso di scrittura, in quella directory può soltanto cancellare i file di sua proprietà. Questo impedisce agli utenti di sovrascrivere o cancellare inavvertitamente i file di qualcun'altro nelle directory ad accesso pubblico, come, ad esempio, `/tmp`.

-O

siete il proprietario del file

-G

l'id di gruppo del file è uguale al vostro

-N

il file è stato modificato dall'ultima volta che è stato letto

f1 -nt f2

il file *f1* è più recente del file *f2*

f1 -ot f2

il file *f1* è meno recente del file *f2*

f1 -ef f2

i file *f1* e *f2* sono hard link allo stesso file

!

"not" -- inverte il risultato delle precedenti opzioni di verifica (restituisce vero se la condizione è assente).

Esempio 7-4. Ricerca di link interrotti (broken link)

```
  1 #!/bin/bash
  2 # broken-link.sh
  3 # Scritto da Lee Bigelow <ligelowbee@yahoo.com>
  4 # Utilizzato con il consenso dell'autore.
  5
  6 #  Uno script di pura shell per cercare i link simbolici "morti" e
visualizzarli
```

```
  7 #+ tra virgolette, in modo tale che possano essere trattati e dati in pasto
a
  8 #+ xargs  :)    es. broken-link.sh /unadirectory  /altradirectory | xargs rm
  9 #
 10 #Il seguente, tuttavia, è il metodo migliore:
 11 #
 12 #find "unadirectory" -type l -print0|\
 13 #xargs -r0 file|\
 14 #grep "broken symbolic"|
 15 #sed -e 's/^\|: *broken symbolic.*$/"/g'
 16 #
 17 #ma non sarebbe bash pura, come deve essere.
 18 #Prudenza: state attenti al file di sistema /proc e a tutti i link
circolari!

19 ################################################################################
 20
 21
 22 #  Se nessun argomento viene passato allo script, la directory di ricerca
 23 #+ directorys viene impostata alla directory corrente. Altrimenti directorys
 24 #+ viene impostata all'argomento passato.
 25 ########################################
 26 [ $# -eq 0 ] && directorys=`pwd` || directorys=$@
 27
 28 #  Implementazione della funzione verlink per cercare, nella directory
 29 #  passatale, i file che sono link a file inesistenti, quindi visualizzarli
 30 #+ tra virgolette. Se uno degli elementi della directory è una
sottodirectory,
 31 #+ allora anche questa viene passata alla funzione verlink.
 32 ##########
 33 verlink () {
 34     for elemento in $1/*; do
 35     [ -h "$elemento" -a ! -e "$elemento" ] && echo \"$elemento\"
 36     [ -d "$elemento" ] && verlink $elemento
 37     # Naturalmente, '-h' verifica i link simbolici, '-d' le directory.
 38     done
 39 }
 40
 41 #  Invia ogni argomento passato allo script alla funzione verlink, se è una
 42 #+ directory valida. Altrimenti viene visualizzato un messaggio d'errore e
le
 43 #+ informazioni sull'utilizzo.
 44 ############################
 45 for directory in $directorys; do
 46     if [ -d $directory ]
 47     then verlink $directory
 48     else
 49         echo "$directory non è una directory"
 50         echo "Utilizzo: $0 dir1 dir2 ..."
 51     fi
 52 done
 53
 54 exit 0
```

Vedi anche Esempio 29-1, Esempio 10-7, Esempio 10-3, Esempio 29-3 e Esempio A-2 che illustrano gli utilizzi degli operatori di verifica di file.

Note

[1] Fate attenzione che il bit *suid* impostato su file binari (eseguibili) può aprire falle di sicurezza

e che il bit *suid* non ha alcun effetto sugli script di shell.

[2] Nei moderni sistemi UNIX, lo sticky bit viene utilizzato solo sulle directory e non più sui file.

Altri operatori di confronto

Un operatore di verifica *binario* confronta due variabili o due grandezze. Si faccia attenzione alla differenza tra il confronto di interi e quello di stringhe.

confronto di interi

-eq

è uguale a

```
if [ "$a" -eq "$b" ]
```

-ne

è diverso (non uguale) da

```
if [ "$a" -ne "$b" ]
```

-gt

è maggiore di

```
if [ "$a" -gt "$b" ]
```

-ge

è maggiore di o uguale a

```
if [ "$a" -ge "$b" ]
```

-lt

è minore di

```
if [ "$a" -lt "$b" ]
```

-le

è minore di o uguale a

```
if [ "$a" -le "$b" ]
```

<

è minore di (tra doppie parentesi)

(("$a" < "$b"))

<=

è minore di o uguale a (tra doppie parentesi)

(("$a" <= "$b"))

>

è maggiore di (tra doppie parentesi)

(("$a" > "$b"))

>=

è maggiore di o uguale a (tra doppie parentesi)

(("$a" >= "$b"))

confronto di stringhe

=

è uguale a

if ["$a" = "$b"]

==

è uguale a

if ["$a" == "$b"]

È sinonimo di =.


Il comportamento dell'operatore di confronto == all'interno del costrutto di verifica doppie parentesi quadre è diverso rispetto a quello nel costrutto parentesi quadre singole.

```
  1 [[ $a == z* ]]    # Vero se $a inizia con una "z"
(corrispondenza di modello).
  2 [[ $a == "z*" ]]  # Vero se $a è uguale a z*
(corrispondenza letterale).
  3
  4 [ $a == z* ]      # Esegue il globbing e la divisione
delle parole.
  5 [ "$a" == "z*" ]  # Vero se $a è uguale a z*
(corrispondenza letterale).
  6
  7 # Grazie a Stephane Chazelas
```

!=

è diverso (non uguale) da

`if [ "$a" != "$b" ]`

Questo operatore esegue la ricerca di corrispondenza all'interno del costrutto [[...]].

<

è inferiore a, in ordine alfabetico ASCII

`if [[ "$a" < "$b" ]]`

`if [ "$a" \< "$b" ]`

Si noti che "<" necessita dell'escaping nel costrutto **`[ ]`**.

>

è maggiore di, in ordine alfabetico ASCII

`if [[ "$a" > "$b" ]]`

`if [ "$a" \> "$b" ]`

Si noti che ">" necessita dell'escaping nel costrutto **`[ ]`**.

Vedi Esempio 26-11 per un'applicazione di questo operatore di confronto.

-z

la stringa è "nulla", cioè, ha lunghezza zero

-n

la stringa non è "nulla".

L'operatore **`-n`** richiede assolutamente il quoting della stringa all'interno delle parentesi quadre. L'utilizzo, tra le parentesi quadre, di una stringa senza quoting sia con **`! -z`**, che da sola (vedi Esempio 7-6) normalmente funziona, tuttavia non è una pratica sicura. Bisogna *sempre* utilizzare il quoting su una stringa da verificare. [1]

Esempio 7-5. Confronti aritmetici e di stringhe

```
 1 #!/bin/bash
 2
 3 a=4
 4 b=5
 5
 6 #  Qui "a" e "b" possono essere trattate sia come interi che come stringhe.
 7 #  Ci si può facilmente confondere tra i confronti aritmetici e quelli sulle
 8 #+ stringhe, perché le variabili Bash non sono tipizzate.
 9 #
10 #  Bash consente le operazioni di interi e il confronto di variabili
```

```
 11 #+ il cui valore è composto solamente da cifre.
 12 #  Attenzione, siete avvisati.
 13
 14 echo
 15
 16 if [ "$a" -ne "$b" ]
 17 then
 18   echo "$a non è uguale a $b"
 19   echo "(confronto aritmetico)"
 20 fi
 21
 22 echo
 23
 24 if [ "$a" != "$b" ]
 25 then
 26   echo "$a non è uguale a $b."
 27   echo "(confronto di stringhe)"
 28   #     "4"  != "5"
 29   # ASCII 52 != ASCII 53
 30 fi
 31
 32 # In questo particolare esempio funziona sia "-ne" che "!=".
 33
 34 echo
 35
 36 exit 0
```

Esempio 7-6. Verificare se una stringa è *nulla*

```
  1 #!/bin/bash
  2 # str-test.sh: Verifica di stringhe nulle e di stringhe senza quoting (*)
  3
  4 # Utilizzando   if [ ... ]
  5
  6
  7 # Se una stringa non è stata inizializzata, non ha un valore definito.
  8 # Questo stato si dice "nullo" (non zero!).
  9
 10 if [ -n $stringa1 ]    # $stringa1 non è stata dichiarata o inizializzata.
 11 then
 12   echo "La stringa \"stringa1\" non è nulla."
 13 else
 14   echo "La stringa \"stringa1\" è nulla."
 15 fi
 16
 17 # Risultato sbagliato.
 18 # Viene visualizzato $stringa1 come non nulla, anche se non era
inizializzata.
 19
 20 echo
 21
 22
 23 # Proviamo ancora.
 24
 25 if [ -n "$stringa1" ]  # Questa volta è stato applicato il quoting a
$stringa1.
 26 then
 27   echo "la stringa \"stringa1\" non è nulla."
 28 else
 29   echo "La stringa \"stringa1\" è nulla."
 30 fi                     #  Usate il quoting per le stringhe nel costrutto
```

```
 31                       #+ di verifica parentesi quadre!
 32
 33
 34 echo
 35
 36
 37 if [ $stringa1 ]       # Qui, $stringa1 è sola.
 38 then
 39   echo "La stringa \"stringa1\" non è nulla."
 40 else
 41   echo "La stringa \"stringa1\" è nulla."
 42 fi
 43
 44 #  Questo funziona bene.
 45 #  L'operatore di verifica [ ] da solo è in grado di rilevare se la stringa
 46 #+ è nulla.
 47 #  Tuttavia è buona pratica usare il quoting ("$stringa1").
 48 #
 49 # Come ha evidenziato Stephane Chazelas,
 50 #    if [ $stringa1 ]    ha un argomento, "]"
 51 #    if [ "$stringa1" ]  ha due argomenti, la stringa vuota "$stringa1" e
"]"
 52
 53
 54 echo
 55
 56
 57
 58 stringa1=inizializzata
 59
 60 if [ $stringa1 ]       # Ancora, $stringa1 da sola.
 61 then
 62   echo "La stringa \"stringa1\" non è nulla."
 63 else
 64   echo "La stringa \"stringa1\" è nulla."
 65 fi
 66
 67 # Ancora, risultato corretto.
 68 # Nondimeno, è meglio utilizzare il quoting ("$stringa1"), perché. . .
 69
 70 stringa1="a = b"
 71
 72 if [ $stringa1 ]       # Ancora $stringa1 da sola.
 73 then
 74   echo "La stringa \"stringa1\" non è nulla."
 75 else
 76   echo "La stringa \"stringa1\" è nulla."
 77 fi
 78
 79 # Senza il quoting di "$stringa1" ora si ottiene un risultato sbagliato!
 80
 81 exit 0
 82
 83 # Grazie anche a Florian Wisser per la "citazione iniziale".
 84
 85 # (*) L'intestazione di commento originaria recita "Testing null strings
 86 #+ and unquoted strings, but not strings and sealing wax, not to
 87 # mention cabbages and kings ..." attribuita a Florian Wisser. La
 88 # seconda riga non è stata tradotta in quanto, la sua traduzione
 89 # letterale, non avrebbe avuto alcun senso nel contesto attuale
 90 # (N.d.T.).
```

Esempio 7-7. zmost

```
 1 #!/bin/bash
 2
 3 # Visualizza i file gzip con  'most'
 4
 5 NOARG=65
 6 NONTROVATO=66
 7 NONGZIP=67
 8
 9 if [ $# -eq 0 ] # stesso risultato di:  if [ -z "$1" ]
10 # $1 può esserci, ma essere vuota:  zmost "" arg2 arg3
11 then
12   echo "Utilizzo: `basename $0` nomefile" >&2
13   # Messaggio d'errore allo stderr.
14   exit $NOARG
15   # Restituisce 65 come exit status dello script (codice d'errore).
16 fi
17
18 nomefile=$1
19
20 if [ ! -f "$nomefile" ]   # Il quoting di $nomefile mantiene gli spazi.
21 then
22   echo "File $nomefile non trovato!" >&2
23   # Messaggio d'errore allo stderr.
24   exit $NONTROVATO
25 fi
26
27 if [ ${nomefile##*.} != "gz" ]
28 # Uso delle parentesi graffe nella sostituzione di variabile.
29 then
30   echo "Il file $1 non è un file gzip!"
31   exit $NONGZIP
32 fi
33
34 zcat $1 | most
35
36 # Usa il visualizzatore di file  'most' (simile a 'less').
37 # Le versioni più recenti di 'most' hanno la capacità di decomprimere un
file.
38 # Si può sostituire con 'more' o 'less', se si desidera.
39
40
41 exit $?   # Lo script restituisce l'exit status della pipe.
42 # In questo punto dello script  "exit $?" è inutile perché lo script,
43 # in ogni caso, restituirà l'exit status dell'ultimo comando eseguito.
```

confronti composti

-a

and logico

exp1 -a exp2 restituisce vero se *entrambe* exp1 e exp2 sono vere.

-o

or logico

exp1 -o exp2 restituisce vero se è vera o exp1 *o* exp2.

Sono simili agli operatori di confronto Bash **&&** e ||, utilizzati all'interno delle doppie parentesi quadre.

```
1 [[ condizione1 && condizione2 ]]
```

Gli operatori **-o** e **-a** vengono utilizzati con il comando **test** o all'interno delle parentesi quadre singole.

```
1 if [ "$exp1" -a "$exp2" ]
```

Fate riferimento ad Esempio 8-3 e ad Esempio 26-16 per vedere all'opera gli operatori di confronto composto.

Note

[1] Come sottolinea S.C., in una verifica composta, il quoting di una variabile stringa può non essere sufficiente. **`[ -n "$stringa" -o "$a" = "$b" ]`** potrebbe, con alcune versioni di Bash, provocare un errore se `$stringa` fosse vuota. Il modo per evitarlo è quello di aggiungere un carattere extra alle variabili che potrebbero essere vuote, **`[ "x$stringa" != x -o "x$a" = "x$b" ]`** (le "x" si annullano).

Costrutti condizionali if/then annidati

È possibile annidare i costrutti condizionali **if/then**. Il risultato è lo stesso di quello ottenuto utilizzando l'operatore di confronto composto **&&** visto precedentemente.

```
1 if [ condizione1 ]
2 then
3   if [ condizione2 ]
4   then
5     fa-qualcosa  # Ma solo se sia "condizione1" che "condizione2" sono vere.
6   fi
7 fi
```

Vedi Esempio 35-4 per una dimostrazione dei costrutti condizionali *if/then* annidati.

Test sulla conoscenza delle verifiche

Il file di sistema `xinitrc` viene di solito impiegato, tra l'altro, per mettere in esecuzione il server X. Questo file contiene un certo numero di costrutti *if/then*, come mostra il seguente frammento.

```
1 if [ -f $HOME/.Xclients ]; then
2   exec $HOME/.Xclients
3 elif [ -f /etc/X11/xinit/Xclients ]; then
4   exec /etc/X11/xinit/Xclients
5 else
6     # failsafe settings.  Although we should never get here
```

```
 7      # (we provide fallbacks in Xclients as well) it can't hurt.
 8      xclock -geometry 100x100-5+5 &
 9      xterm -geometry 80x50-50+150 &
10      if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
11              netscape /usr/share/doc/HTML/index.html &
12      fi
13 fi
```

Spiegate i costrutti di "verifica" del frammento precedente, quindi esaminate l'intero file `/etc/X11/xinit/xinitrc` ed analizzate i costrutti *if/then*. È necessario consultare i capitoli riguardanti grep, sed e le espressioni regolari più avanti.

Operazioni ed argomenti correlati

Operatori

assegnamento

`assegnamento di variabile`

Inizializzare o cambiare il valore di una variabile

=

Operatore di assegnamento multiuso, utilizzato sia per gli assegnamenti aritmetici che di stringhe.

```
 1 var=27
 2 categoria=minerali  # Non sono consentiti spazi né prima né dopo
l'"=".
```

Non bisogna assolutamente confondere l'"=" operatore di assegnamento con l'= operatore di verifica.

```
 1 #    = come operatore di verifica
 2
 3 if [ "$stringa1" = "$stringa2" ]
 4 # if [ "X$stringa1" = "X$stringa2" ] è più sicuro, evita
un
 5 #+ messaggio d'errore se una delle variabili dovesse
essere vuota.
 6 # (Le due "X" anteposte si annullano).
 7 then
 8    comando
 9 fi
```

operatori aritmetici

+

più

-

meno

*

per

/

diviso

**

elevamento a potenza

```
  1 # La versione 2.02 di Bash ha introdotto l'operatore di elevamento a
potenza  "**".
  2
  3 let "z=5**3"
  4 echo "z = $z"   # z = 125
```

%

modulo, o mod (restituisce il resto di una divisione tra interi)

```
bash$ echo `expr 5 % 3`
2
```

Questo operatore viene utilizzato, tra l'altro, per generare numeri in un determinato intervallo (vedi Esempio 9-23, Esempio 9-26) e per impaginare l'output dei programmi (vedi Esempio 26-15 e Esempio A-7). È anche utile per generare numeri primi, (vedi Esempio A-17). Modulo si trova sorprendentemente spesso in diverse formule matematiche.

Esempio 8-1. Massimo comun divisore

```
  1 #!/bin/bash
  2 # gcd.sh: massimo comun divisore
  3 #         Uso dell'algoritmo di Euclide
  4
  5 #  Il "massimo comun divisore" (MCD) di due interi è l'intero
  6 #+ più grande che divide esattamente entrambi.
  7
  8 #  L'algoritmo di Euclide si basa su divisioni successive.
  9 #  Ad ogni passaggio,
 10 #+ dividendo <---  divisore
 11 #+ divisore  <---  resto
 12 #+ finché resto = 0.
 13 #+ Nell'ultimo passaggio MCD = dividendo.
 14 #
 15 #  Per un'eccellente disamina dell'algoritmo di Euclide, vedi
 16 #  al sito di Jim Loy, http://www.jimloy.com/number/euclids.htm.
 17
 18
 19 # ------------------------------------------------------
 20 # Verifica degli argomenti
 21 ARG=2
```

```
 22 E_ERR_ARG=65
 23
 24 if [ $# -ne "$ARG" ]
 25 then
 26   echo "Utilizzo: `basename $0` primo-numero secondo-numero"
 27   exit $E_ERR_ARG
 28 fi
 29 # ---------------------------------------------------------
 30
 31
 32 mcd ()
 33 {
 34
 35                                  #  Assegnamento arbitrario.
 36   dividendo=$1                   #  Non ha importanza
 37   divisore=$2                    #+ quale dei due è maggiore.
 38                                  #  Perché?
 39
 40   resto=1                        #  Se la variabile usata in un
ciclo non è
 41                                  #+ inizializzata, il risultato è un
errore
 42                                  #+ al primo passaggio nel ciclo.
 43
 44   until [ "$resto" -eq 0 ]
 45   do
 46     let "resto = $dividendo % $divisore"
 47     dividendo=$divisore          # Ora viene ripetuto con 2 numeri
più piccoli.
 48     divisore=$resto
 49   done                           # Algoritmo di Euclide
 50
 51 }                                # L'ultimo $dividendo è il MCD.
 52
 53
 54 mcd $1 $2
 55
 56 echo; echo "MCD di $1 e $2 = $dividendo"; echo
 57
 58
 59 # Esercizio :
 60 # --------
 61 #  Verificate gli argomenti da riga di comando per essere certi che
siano
 62 #+ degli interi, se non lo fossero uscite dallo script con un
adeguato
 63 #+ messaggio d'errore.
 64
 65 exit 0
```

+=

"più-uguale" (incrementa una variabile con una costante

`let "var += 5"` come risultato `var` è stata incrementata di `5`.

-=

"meno-uguale" (decrementa una variabile di una costante)

*=

"per-uguale" (moltiplica una variabile per una costante)

`let "var *= 4"` come risultato `var` è stata moltiplicata per `4`.

/=

"diviso-uguale" (divide una variabile per una costante)

%=

"modulo-uguale" (resto della divisione di una variabile per una costante)

Gli operatori aritmetici si trovano spesso in espressioni con expr o let.

Esempio 8-2. Utilizzo delle operazioni aritmetiche

```
 1 #!/bin/bash
 2 # Contare fino a 11 in 10 modi diversi.
 3
 4 n=1; echo -n "$n "
 5
 6 let "n = $n + 1"   # Va bene anche let "n = n + 1".
 7 echo -n "$n "
 8
 9
10 : $((n = $n + 1))
11 #  I ":" sono necessari perché altrimenti Bash tenta
12 #+ di interpretare "$((n = $n + 1))" come un comando.
13 echo -n "$n "
14
15 (( n = n + 1 ))
16 #  Alternativa più semplice del metodo precedente.
17 #  Grazie a David Lombard per la precisazione.
18 echo -n "$n "
19
20 n=$(($n + 1))
21 echo -n "$n "
22
23 : $[ n = $n + 1 ]
24 #  I ":" sono necessari perché altrimenti Bash tenta
25 #+ di interpretare "$[ n = $n + 1 ]" come un comando.
26 # Funziona anche se "n" fosse inizializzata come stringa.
27 echo -n "$n "
28
29 n=$[ $n + 1 ]
30 #  Funziona anche se "n" fosse inizializzata come
stringa.
31 #* Evitate questo costrutto perché è obsoleto e non
portabile.
32 # Grazie, Stephane Chazelas.
33 echo -n "$n "
34
35 # Ora con gli operatori di incremento in stile C.
36 # Grazie a Frank Wang per averlo segnalato.
37
38 let "n++"          # anche con let "++n".
39 echo -n "$n "
40
```

```
41 (( n++ ))          # anche con (( ++n ).
42 echo -n "$n "
43
44 : $(( n++ ))       # anche con : $(( ++n )).
45 echo -n "$n "
46
47 : $[ n++ ]         # e anche : $[ ++n ]]
48 echo -n "$n "
49
50 echo
51
52 exit 0
```

In Bash, attualmente, le variabili intere sono del tipo *signed long* (32-bit) comprese nell'intervallo da -2147483648 a 2147483647. Un'operazione comprendente una variabile con un valore al di fuori di questi limiti dà un risultato sbagliato.

```
  1 a=2147483646
  2 echo "a = $a"      # a = 2147483646
  3 let "a+=1"         # Incrementa "a".
  4 echo "a = $a"      # a = 2147483647
  5 let "a+=1"         # incrementa ancora "a", viene oltrepassato
il limite.
  6 echo "a = $a"      # a = -2147483648
  7                    #      ERRORE (fuori intervallo)
```

Dalla versione 2.05b, Bash supporta gli interi di 64 bit.

Bash non contempla l'aritmetica in virgola mobile. Considera i numeri che contengono il punto decimale come stringhe.

```
1 a=1.5
2
3 let "b = $a + 1.3"  # Errore.
4 #  t2.sh: let: b = 1.5 + 1.3: syntax error in expression
5 #+ (error token is ".5 + 1.3")
6
7 echo "b = $b"       # b=1
```

Si utilizzi bc negli script in cui sono necessari calcoli in virgola mobile oppure le librerie di funzioni matematiche.

Operatori bitwise. Gli operatori bitwise compaiono raramente negli script di shell. L'uso principale sembra essere quello di manipolare e verificare i valori letti dalle porte o dai socket. "Lo scorrimento di bit" è più importante nei linguaggi compilati, come il C e il C++, che sono abbastanza veloci per consentirne un uso proficuo.

operatori bitwise

<<

scorrimento a sinistra (moltiplicazione per 2 per ogni posizione spostata)

<<=

"scorrimento a sinistra-uguale"

`let "var <<= 2"` come risultato i bit di `var` sono stati spostati di `2` posizioni verso sinistra (moltiplicazione per `4`)

>>

scorrimento a destra (divisione per `2` per ogni posizione spostata)

>>=

"scorrimento a destra-uguale" (inverso di <<=)

&

AND bitwise

&=

"AND bitwise-uguale"

|

OR bitwise

|=

"OR bitwise-uguale"

~

complemento bitwise

!

NOT bitwise

^

XOR bitwise

^=

"XOR bitwise-uguale"

operatori logici

&&

and (logico)

```
1 if [ $condizione1 ] && [ $condizione2 ]
2 # Uguale a:  if [ $condizione1 -a $condizione2 ]
```

```
  3 # Restituisce vero se entrambe, condizione1 e condizione2, sono
vere...
  4
  5 if [[ $condizione1 && $condizione2 ]]    # Funziona anche così.
  6 # Notate che l'operatore && non è consentito nel costrutto [ ... ].
```

&& può essere utilizzato, secondo il contesto, in una lista and per concatenare dei comandi.

||

or (logico)

```
  1 if [ $condizione1 ] || [ $condizione2 ]
  2 # Uguale a:  if [ $condizione1 -o $condizione2 ]
  3 # Restituisce vero se o condizione1 o condizione2 è vera...
  4
  5 if [[ $condizione1 || $condizione2 ]]    # Funziona anche così.
  6 # Notate che l'operatore || non è consentito nel costrutto [ ... ].
```

Bash verifica l'exit status di ogni enunciato collegato con un operatore logico.

Esempio 8-3. Condizioni di verifica composte utilizzando && e ||

```
  1 #!/bin/bash
  2
  3 a=24
  4 b=47
  5
  6 if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
  7 then
  8   echo "Verifica nr.1 eseguita con successo."
  9 else
 10   echo "Verifica nr.1 fallita."
 11 fi
 12
 13 # ERRORE:  if [ "$a" -eq 24 && "$b" -eq 47 ]
 14 #+         cerca di eseguire  ' [ "$a" -eq 24 '
 15 #+         e fallisce nella ricerca di corrispondenza di ']'.
 16 #
 17 # Nota:   if [[ $a -eq 24 && $b -eq 24 ]]   funziona
 18 #  La verifica if con le doppie parentesi quadre è più flessibile
 19 #+ della versione con le paretesi quadre singole.
 20 #    ("&&" ha un significato diverso nella riga 17 di quello della
riga 6.).
 21 #    Grazie a Stephane Chazelas per averlo evidenziato.
 22
 23
 24 if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
 25 then
 26   echo "Verifica nr.2 eseguita con successo."
 27 else
 28   echo "Verifica nr.2 fallita."
 29 fi
 30
 31
 32 #  Le opzioni -a e -o offrono
 33 #+ una condizione di verifica composta alternativa.
 34 #  Grazie a Patrick Callahan per la precisazione.
 35
```

```
36
37 if [ "$a" -eq 24 -a "$b" -eq 47 ]
38 then
39   echo "Verifica nr.3 eseguita con successo."
40 else
41   echo "Verifica nr.3 fallita."
42 fi
43
44
45 if [ "$a" -eq 98 -o "$b" -eq 47 ]
46 then
47   echo "Verifica nr.4 eseguita con successo."
48 else
49   echo "Verifica nr.4 fallita."
50 fi
51
52
53 a=rinoceronte
54 b=coccodrillo
55 if [ "$a" = rinoceronte ] && [ "$b" = coccodrillo ]
56 then
57   echo "Verifica nr.5 eseguita con successo."
58 else
59   echo "Verifica nr.5 fallita."
60 fi
61
62 exit 0
```

Gli operatori && e || vengono utilizzati anche nel contesto matematico.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

operatori diversi

,

operatore virgola

L'**operatore virgola** concatena due o più operazioni aritmetiche. Vengono valutate tutte le operazioni (con possibili *effetti collaterali*), ma viene restituita solo l'ultima.

```
 1 let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
 2 echo "t1 = $t1"               # t1 = 11
 3
 4 let "t2 = ((a = 9, 15 / 3))"  # Imposta "a" e calcola "t2"
 5 echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

L'operatore virgola viene impiegato principalmente nei cicli for. Vedi Esempio 10-12.

Parte del manuale originale

Costanti numeriche

Lo script di shell interpreta un numero come numero decimale (base 10), tranne quando quel numero è scritto in una notazione particolare: con un prefisso specifico. Un numero preceduto da *`0`* è un numero *`ottale`* (base 8). Un numero preceduto da *`0x`* è un numero *`esadecimale`* (base 16). Un numero contenente un *`#`* viene valutato come *`BASE#NUMERO`* (con limitazioni di notazione ed ampiezza).

Esempio 8-4. Rappresentazione di costanti numeriche

```
 1 #!/bin/bash
 2 # numbers.sh: Rappresentazione di numeri con basi differenti.
 3
 4 # Decimale: quella preimpostata
 5 let "dec = 32"
 6 echo "numero decimale = $dec"             # 32
 7 # Qui non c'è niente di insolito.
 8
 9
10 # Ottale: numeri preceduti da '0' (zero)
11 let "oct = 032"
12 echo "numero ottale = $ott"               # 26
13 # Risultato visualizzato come decimale.
14 # --------- ------ -- ----------------
15
16 # Esadecimale: numeri preceduti da '0x' o '0X'
17 let "esa = 0x32"
18 echo "numero esadecimale = $esa"          # 50
19 # Risultato visualizzato come decimale.
20
21 #  Altre basi: BASE#NUMERO
22 #  BASE tra 2 e 64.
23 #  NUMERO deve essere formato dai simboli nell'intervallo indicato da
24 #+ BASE, vedi di seguito.
25
26 let "bin = 2#111100111001101"
27 echo "numero binario = $bin"              # 31181
28
29 let "b32 = 32#77"
30 echo "numero in base 32 = $b32"           # 231
31
32 let "b64 = 64#@_"
33 echo "numero in base 64 = $b64"           # 4031
34 #
35 # Questa notazione funziona solo per un intervallo limitato (2 - 64)
36 # 10 cifre + 26 caratteri minuscoli + 26 caratteri maiuscoli + @ + _
37
38 echo
39
40 echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
41                                             # 1295 170 44822 3375
42
43
44 #  Nota importante:
45 #  --------------
46 #  Utilizzare un simbolo al di fuori dell'intervallo della base specificata
47 #+ provoca un messaggio d'errore.
48
49 let "ott_errato = 081"
50 # Messaggio d'errore visualizzato:
51 #  numbers.sh: let: ott_errato = 081: value too great for base
52 #+ (error token is "081")
```

```
53 #             I numeri ottali utilizzano solo cifre nell'intervallo 0 - 7.
54
55 exit 0       # Grazie, Rich Bartell e Stephane Chazelas, per il chiarimento.
```

Qui abbiamo potuto riassumere ben poco anche perché ci sono degli argomenti che sono già semplificati dall’autore traduttore, ma è già qualcosa che è in italiano. Buona Lettura..