Corso facilitato di bash scripting

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

I contenuti di questa guida della (lezione Parte quarta) sono riportati in originale (contenuto indispensabile con esempi pratici). I link sono cliccabili, utili per raggiungre il sito ufficiale dal web. Globalmente la guida è conclusa qui ma riporterò altre sezioni di riepilogo.

Parte 4. Argomenti avanzati

Giunti a questo punto, si è pronti a sviscerare alcuni degli aspetti più difficili ed insoliti dello scripting. Strada facendo, si cercherà di "andare oltre le proprie capacità" in vari modi e di esaminare *condizioni limite* (cosa succede quando ci si deve muovere in un territorio sconosciuto senza una cartina?).

Sommario

Capitolo 19. Espressioni Regolari

Sommario

Per sfruttare pienamente la potenza dello scripting di shell, occorre conoscere a fondo le Espressioni Regolari. Alcune utility e comandi comunemente impiegati negli script, come grep, expr, sed e awk, interpretano ed usano le ER.

19.1. Una breve introduzione alle Espressioni Regolari

Un'espressione è una stringa di caratteri. Quei caratteri che hanno un'interpretazione che va al di là del loro significato letterale vengono chiamati *metacaratteri*. Le virgolette, ad esempio, possono indicare la frase di una persona in un dialogo, *idem* o il meta-significato dei simboli che seguono. Le Espressioni Regolari sono serie di caratteri e/o metacaratteri a cui un sistema operativo attribuisce funzionalità speciali. [1]

Le Espressioni Regolari (ER) vengono principalmente impiegate nelle ricerche di un testo e nella manipolazione di stringhe. Una ER *verifica* un singolo carattere o una serie di caratteri -- una sottostringa o una stringa intera.

- L'asterisco -- * -- verifica un numero qualsiasi di ripetizioni della stringa di caratteri o l'ER che lo precede, compreso *nessun carattere*.

  "1133*" verifica *`11 + uno o più 3 + altri possibili caratteri`*: *`113`*, *`1133`*, *`111312`*, eccetera.

- Il punto -- . -- verifica un carattere qualsiasi , tranne il ritorno a capo. [2]

  "13." verifica *`13 + almeno un carattere qualsiasi (compreso lo spazio)`*: *`1133`*, *`11333`*, ma non il solo *`13`*.

- L'accento circonflesso -- ^ -- verifica l'inizio di una riga, ma talvolta, secondo il contesto, nega il significato della serie di caratteri in una ER.
- Il simbolo del dollaro -- $ -- alla fine di una ER verifica la fine di una riga.

  "^$" verifica le righe vuote.

  Sia ^ che $ sono chiamate *àncore*, poichè indicano, àncorano, una posizione all'interno di una ER.

- Le parantesi quadre -- [...] -- racchiudono una serie di caratteri da verificare in una singola ER.

  "[xyz]" verifica i caratteri *x*, *y* o *z*.

  "[c-n]" verifica tutti i caratteri compresi nell'intervallo da *c* a *n*.

  "[B-Pk-y]" verifica tutti i caratteri compresi negli intervalli da *B* a *P* e da *k* a *y*.

  "[a-z0-9]" verifica tutte le lettere minuscole e/o tutte le cifre.

  "[^b-d]" verifica tutti i caratteri *tranne* quelli compresi nell'intervallo da *b* a *d*. Questo è un esempio di ^ che nega, o inverte, il significato della ER che segue (assumendo un ruolo simile a ! in un contesto diverso).

  Sequenze combinate di caratteri racchiusi tra parentesi quadre verificano le possibili modalità di scrittura di una parola. "[Yy][Ee][Ss]" verifica *`yes`*, *`Yes`*, *`YES`*, *`yEs`* e così via. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" verifica il numero di Previdenza Sociale. (negli U.S.A [N.d.T.]).

- La barra inversa -- \ -- è il carattere di escape per un carattere speciale, il che significa che quel carattere verrà interpretato letteralmente.

  "\$" riporta il simbolo del "$" al suo significato letterale, invece che a quello di fine riga in una ER. Allo stesso modo "\\" assume il significato letterale di "\".

- "Parentesi acute" con escaping -- \<...\> -- indicano l'inizio e la fine di una parola.

  Le parentesi acute vanno usate con l'escaping perché, altrimenti, avrebbero il loro significato letterale.

  "\<tre\>" verifica la parola "tre", ma non "treno", "otre", "strega", ecc.

```
bash$ cat filetesto
Questa è la riga 1, che è unica.
Questa è l'unica riga 2.
Questa è la riga 3, un'altra riga.
Questa è la riga 4.


bash$ grep 'un' filetesto
Questa è la riga 1, che è unica.
Questa è l'unica riga 2.
Questa è la riga 3, un'altra riga.


bash$ grep '\<un\>' filetesto
Questa è la riga 3, un'altra riga.
```

Il solo modo per essere certi che una ER particolare funzioni è provare.

```
 1 FILE DI PROVA: tstfile                          # Non verificato.
 2                                                 # Non verificato.
 3 Eseguite  grep "1133*" su questo file.          # Verificato.
 4                                                 # Non verificato.
 5                                                 # Non verificato.
 6 Questa riga contiene il numero 113.             # Verificato
 7 Questa riga contiene il numero 13.              # Non verificato.
 8 Questa riga contiene il numero 133.             # Non verificato.
 9 Questa riga contiene il numero 1133.            # Verificato.
10 Questa riga contiene il numero 113312.          # Verificato.
11 Questa riga contiene il numero 1112.            # Non verificato.
12 Questa riga contiene il numero 113312312.       # Verificato.
13 Questa riga non contiene alcun numero.          # Non verificato.
```

```
bash$ grep "1133*" tstfile
Eseguite  grep "1133*" su questo file.      # Verificato .
Questa riga contiene il numero 113.         # Verificato.
Questa riga contiene il numero 1133.        # Verificato.
Questa riga contiene il numero 113312.      # Verificato.
Questa riga contiene il numero 113312312.   # Verificato.
```

- **ER estese.** Usate con egrep, awk e Perl
- Il punto interrogativo -- ? -- verifica uno o nessun carattere dell'ER che lo precede. Viene generalmente usato per verificare singoli caratteri.

- Il più -- + -- verifica uno o più caratteri della ER che lo precede. Svolge un ruolo simile all'*, ma *non* verifica l'occorrenza zero (nessuna occorrenza).

```
 1 # La versione GNU di sed e awk può usare "+",
 2 # ma è necessario l'escaping.
 3
 4 echo a111b | sed -ne '/a1\+b/p'
 5 echo a111b | grep 'a1\+b'
 6 echo a111b | gawk '/a1+b/'
 7 # Tutte queste forme si equivalgono.
 8
 9 # Grazie, S.C.
```

- "Parentesi graffe" con escaping -- \{\} -- indicano il numero di occorrenze da verificare nella ER che le precede.

  L'escaping delle parentesi graffe è necessario perché, altrimenti, avrebbero semplicemente il loro significato letterale. Quest'uso, tecnicamente, non fa parte della serie di ER di base.

  "[0-9]\{5\}" verifica esattamente cinque cifre (nell'intervallo da 0 a 9).

  Le parentesi graffe non sono disponibili come ER nella versione "classica" (non-POSIX compliant) di awk. Comunque, **gawk** possiede l'opzione `--re-interval` che le consente (senza dover usare l'escaping).

  ```
  bash$ echo 2222 | gawk --re-interval '/2{3}/'
  2222
  ```

  Perl ed alcune versioni di **egrep** non richiedono l'escaping delle parentesi graffe.

- Parentesi -- **( )** -- racchiudono gruppi di ER. Sono utili seguite dall'operatore "|" e nelle estrazioni di sottostringa che usano expr.
- -- | -- l'operatore "or" delle ER verifica una serie qualsiasi di caratteri alternativi.

```
bash$ egrep '(l|r)egge' misc.txt
La persona che legge sembra essere meglio informata di chi non lo fa.
Il re saggio regge il proprio regno con giustizia.
```

Alcune versioni di **sed**, **ed** e **ex** supportano le versioni con escaping delle Espressioni Regolari estese descritte prima allo stesso modo delle utility GNU

- **Classi di caratteri POSIX. `[:classe:]`**

  Rappresentano un metodo alternativo per specificare un intervallo di caratteri da verificare.

- **`[:alnum:]`** verifica i caratteri alfabetici e/o numerici. Equivale a **`A-Za-z0-9`**.
- **`[:alpha:]`** verifica i caratteri alfabetici. Equivale a **`A-Za-z`**.
- **`[:blank:]`** verifica uno spazio o un carattere di tabulazione.
- **`[:cntrl:]`** verifica i caratteri di controllo.
- **`[:digit:]`** verifica le cifre (decimali). Equivale a **`0-9`**.

- **[:graph:]** (caratteri grafici stampabili). Verifica i caratteri nell'intervallo 33 - 126 della codifica ASCII. È uguale a **[:print:]**, vedi oltre, ma esclude il carattere di spazio.
- **[:lower:]** verifica i caratteri alfabetici minuscoli. Equivale a **a-z**.
- **[:print:]** (caratteri stampabili). Verifica i caratteri nell'intervallo 32 - 126 della codifica ASCII. È uguale a **[:graph:]**, visto prima, ma con l'aggiunta del carattere di spazio.
- **[:space:]** verifica i caratteri di spaziatura (spazio e tabulazione orizzontale).
- **[:upper:]** verifica i caratteri alfabetici maiuscoli. Equivale a **A-Z**.
- **[:xdigit:]** verifica le cifre esadecimali. Equivale a **0-9A-Fa-f**.

Le classi di caratteri POSIX generalmente richiedono il quoting o le doppie parentesi quadre ([[ ]]).

```
bash$ grep [[:digit:]] fileprova
abc=723
```

Queste classi di caratteri possono anche essere usate con il globbing, sebbene limitatamente.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

Per vedere le classi di caratteri POSIX negli script, si faccia riferimento all'Esempio 12-17 e Esempio 12-18.

Sed, awk e Perl, usati come filtri negli script, trattano le ER come argomenti, quando devono "vagliare" o trasformare file o flussi di I/O. Vedi Esempio A-13 e Esempio A-18 per una descrizione di questa funzionalità.

Il riferimento fondamentale per questo complesso argomento è "Mastering Regular Expressions" di Friedl. Anche "Sed & Awk"," di Dougherty e Robbins fornisce una lucidissima trattazione delle ER. Vedi *Bibliografia* per ulteriori informazioni su questi libri.

Note

[1] Il tipo più semplice di Espressione Regolare è una stringa di caratteri con il suo solo significato letterale, che non contiene, quindi, nessun metacarattere.

[2] Poiché sed, awk e grep elaborano le righe, di solito non ci sarà bisogno di verificare un ritorno a capo. In quei casi in cui dovesse esserci un ritorno a capo, perché inserito in una espressione su più righe, il punto lo verifica.

```
1 #!/bin/bash
2
3 sed -e 'N;s/.*/[&]/' << EOF   # Here document
4 riga1
5 riga2
6 EOF
7 # OUTPUT:
8 # [riga1
9 # riga2]
```

```
10
11
12
13 echo
14
15 awk '{ $0=$1 "\n" $2; if (/riga.1/) {print}}' << EOF
16 riga 1
17 riga 2
18 EOF
19 # OUTPUT:
20 # riga
21 # 1
22
23
24 # Grazie, S.C.
25
26 exit 0
```

19.2. Globbing

Bash, di per sé, non è in grado di riconoscere le Espressioni Regolari. Negli script, sono i comandi e le utility , come sed e awk, che interpretano le ER.

Bash, invece, esegue l'espansione del nome dei file, un processo conosciuto come "globbing" che, però, *non* usa la serie standard di caratteri delle ER, ma riconosce ed espande i caratteri jolly. Il globbing interpreta i caratteri jolly standard * e ?, liste di caratteri racchiuse tra parentesi quadre ed alcuni altri caratteri speciali (come ^, che nega il senso di una ricerca). Esistono, tuttavia, alcune importanti limitazioni nell'impiego dei caratteri jolly. Stringhe che contengono l'`*` non verificano i nomi dei file che iniziano con un punto, come, ad esempio, `.bashrc`. [1] In modo analogo, il *?* ha un significato diverso da quello che avrebbe se impiegato in una ER.

```
bash$ ls -l
total 2
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt
```

```
bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt
```

Bash esegue l'espanione del nome del file sugli argomenti passati da riga di comando senza il quoting. Il comando echo dimostra questa funzionalità.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```


È possibile modificare il modo in cui Bash interpreta i caratteri speciali nel globbing. Il comando **set -f** disabilita il globbing e shopt, con le opzioni `nocaseglob` e `nullglob`, ne muta il comportamento.

Vedi anche Esempio 10-4.

Note

[1] L'espansione del nome del file *può* verificare i nomi di file che iniziano con il punto, ma solo se il modello lo include esplicitamente.

```
  1 ~/[.]bashrc    #  Non viene espanso a ~/.bashrc
  2 ~/?bashrc      #  Neanche in questo caso.
  3                #  Nel globbing i caratteri jolly e i metacaratteri NON
  4                #+ espandono il punto.
  5
  6 ~/.[b]ashrc    #  Viene espanso a ~./bashrc
  7 ~/.ba?hrc      #  Idem.
  8 ~/.bashr*      #  Idem.
  9
 10 #  Impostando l'opzione "dotglob" si abilita anche l'espansione del
punto.
 11
 12 #  Grazie, S.C.
```

Capitolo 20. Subshell

Quando si esegue uno script viene lanciata un'altra istanza del processore dei comandi. Proprio come i comandi vengono interpretati al prompt della riga di comando, così fa uno script che deve elaborarne una lista. Ogni script di shell in esecuzione è, in realtà, un sottoprocesso della shell genitore, quella che fornisce il prompt alla console o in una finestra di xterm.

Anche uno script di shell può mettere in esecuzione dei sottoprocessi. Queste *subshell* consentono allo script l'elaborazione in parallelo, vale a dire, l'esecuzione simultanea di più compiti di livello inferiore.

Di solito, un comando esterno presente in uno script genera un sottoprocesso, al contrario di un builtin di Bash. È per questa ragione che i builtin vengono eseguiti più velocemente dei loro equivalenti comandi esterni.

Elenco di comandi tra parentesi

(comando1; comando2; comando3; ...)

Una lista di comandi tra *parentesi* dà luogo ad una subshell.

Le variabili presenti in una subshell *non* sono visibili al di fuori del suo blocco di codice. Non sono accessibili al processo genitore, quello che ha lanciato la subshell. Sono, a tutti gli effetti, variabili locali.

Esempio 20-1. Ambito di una variabile in una subshell

```
 1 #!/bin/bash
 2 # subshell.sh
 3
 4 echo
 5
 6 echo "Livello della subshell all'ESTERNO della subshell = $BASH_SUBSHELL"
 7 # Bash, versione 3, adotta la nuova variabile             $BASH_SUBSHELL.
 8 echo
 9
10 variabile_esterna=Esterna
11
12 (
13 echo "Livello della subshell all'INTERNO della subshell = $BASH_SUBSHELL"
14 variabile_interna=Interna
15 echo "Nella subshell, \"variabile_interna\" = $variabile_interna"
16 echo "Nella subshell, \"variabile_esterna\" = $variabile_esterna"
17 )
18
19 echo
20 echo "Livello della subshell all'ESTERNO della subshell = $BASH_SUBSHELL"
21 echo
22
23 if [ -z "$variabile_interna" ]
24 then
25   echo "variabile_interna non definita nel corpo principale della shell"
26 else
27   echo "variabile_interna definita nel corpo principale della shell"
28 fi
29
30 echo "Nel corpo principale della shell,\
31 \"variabile_interna\" = $variabile_interna"
32 # $variabile_interna viene indicata come non inizializzata perché
33 # le variabili definite in una subshell sono "variabili locali".
34
35 echo
36
37 exit 0
```

Vedi anche Esempio 32-2.

+

I cambiamenti di directory effettuati in una subshell non si ripercuotono sulla shell genitore.

Esempio 20-2. Elenco dei profili utente

```
 1 #!/bin/bash
 2 # allprofs.sh: visualizza i profili di tutti gli utenti
 3
 4 # Script di Heiner Steven modificato dall'autore di questo documento.
 5
 6 FILE=.bashrc  #  Il file contenente il profilo utente
 7               #+ nello script originale era ".profile".
 8
 9 for home in `awk -F: '{print $6}' /etc/passwd`
10 do
11   [ -d "$home" ] || continue    #  Se non vi è la directory home,
12                                 #+ va al successivo.
13   [ -r "$home" ] || continue    #  Se non ha i permessi di lettura, va
14                                 #+ al successivo.
15
16   (cd $home; [ -e $FILE ] && less $FILE)
17 done
18
19 #  Quando lo script termina, non è necessario un 'cd' alla directory
20 #+ originaria, perché 'cd $home' è stato eseguito in una subshell.
21
22 exit 0
```

Una subshell può essere usata per impostare un "ambiente dedicato" per un gruppo di comandi.

```
 1 COMANDO1
 2 COMANDO2
 3 COMANDO3
 4 (
 5   IFS=:
 6   PATH=/bin
 7   unset TERMINFO
 8   set -C
 9   shift 5
10   COMANDO4
11   COMANDO5
12   exit 3 # Esce solo dalla subshell.
13 )
14 # La shell genitore non è stata toccata ed il suo ambiente è preservato.
15 COMANDO6
16 COMANDO7
```

Una sua applicazione permette di verificare se una variabile è stata definita.

```
 1 if (set -u; : $variabile) 2> /dev/null
 2 then
 3   echo "La variabile è impostata."
 4 fi     #  La variabile potrebbe essere stata impostata nello script stesso,
 5        #+ oppure essere una variabile interna di Bash,
 6        #+ oppure trattarsi di una variabile d'ambiente (che è stata
esportata).
 7
 8 #  Si sarebbe anche potuto scrivere
```

```
 9 #                   [[ ${variabile-x} != x || ${variabile-y} !=y ]]
10 # oppure            [[ ${variabile-x} != x$variabile ]]
11 # oppure            [[ ${variabile+x} = x ]]
12 # oppure            [[ ${variabile-x} != x ]]
```

Un'altra applicazione è quella di verificare la presenza di un file lock:

```
 1 if (set -C; : > file_lock) 2> /dev/null
 2 then
 3   :   # il file_lock non esiste: nessun utente sta eseguendo lo script
 4 else
 5   echo "C'è già un altro utente che sta eseguendo quello script."
 6 exit 65
 7 fi
 8
 9 #  Frammento di codice di Stephane Chazelas,
10 #+ con modifiche effettuate da Paulo Marcel Coelho Aragao.
```

È possibile eseguire processi in parallelo per mezzo di differenti subshell. Questo permette di suddividere un compito complesso in sottocomponenti elaborate contemporaneamente.

Esempio 20-3. Eseguire processi paralleli con le subshell

```
 1         (cat lista1 lista2 lista3 | sort | uniq > lista123) &
 2         (cat lista4 lista5 lista6 | sort | uniq > lista456) &
 3         # Unisce ed ordina entrambe le serie di liste simultaneamente.
 4         # L'esecuzione in background assicura l'esecuzione parallela.
 5         #
 6         # Stesso effetto di
 7         #   cat lista1 lista2 lista3 | sort | uniq > lista123 &
 8         #   cat lista4 lista5 lista6 | sort | uniq > lista456 &
 9
10         wait   #  Il comando successivo non viene eseguito finché le
subshell
11                #+ non sono terminate.
12
13         diff lista123 lista456
```

Per la redirezione I/O a una subshell si utilizza l'operatore di pipe "|", come in **`ls -al | (comando)`**.

Un elenco di comandi tra *`parentesi graffe`* *non* esegue una subshell.

{ comando1; comando2; comando3; ... }

Capitolo 21. Shell con funzionalità limitate.

Azioni disabilitate in una shell ristretta

L'esecuzione di uno script, o di una parte di uno script, in *modalità ristretta* impedisce l'esecuzione di alcuni comandi normalmente disponibili. Rappresenta una misura di sicurezza per limitare i privilegi dell'utilizzatore dello script e per minimizzare possibili danni causati dalla sua esecuzione.

Usare *cd* per modificare la directory di lavoro.

Cambiare i valori delle variabili d'ambiente *$PATH*, *$SHELL*, *$BASH_ENV*, o *$ENV*.

Leggere o modificare *$SHELLOPTS*, le opzioni delle variabili d'ambiente di shell.

Redirigere l'output.

Invocare comandi contenenti una o più /.

Invocare *exec* per sostituire la shell con un processo differente.

Diversi altri comandi che consentirebbero o un uso maldestro o tentativi per sovvertire lo script a finalità per le quali non era stato progettato.

Uscire dalla modalità ristretta dall'interno dello script.

Esempio 21-1. Eseguire uno script in modalità ristretta

```
 1 #!/bin/bash
 2 # Far iniziare lo script con "#!/bin/bash -r"
 3 # significa eseguire l'intero script in modalità ristretta.
 4
 5 echo
 6
 7 echo "Cambio di directory."
 8 cd /usr/local
 9 echo "Ora ti trovi in `pwd`"
10 echo "Ritorno alla directory home."
11 cd
12 echo "Ora ti trovi in `pwd`"
13 echo
14
15 # Quello fatto fin qui è normale, modalità non ristretta.
16
17 set -r
18 # set --restricted    ha lo stesso significato.
19 echo "==> Ora lo script è in modalità ristretta. <=="
20
21 echo
22 echo
23
24 echo "Tentativo di cambiamento di directory in modalità ristretta."
25 cd ..
26 echo "Ti trovi ancora in `pwd`"
27
28 echo
29 echo
30
31 echo "\$SHELL = $SHELL"
32 echo "Tentativo di cambiare la shell in modalità ristretta."
33 SHELL="/bin/ash"
34 echo
35 echo "\$SHELL= $SHELL"
36
37 echo
38 echo
39
```

```
40 echo "Tentativo di redirigere l'output in modalità ristretta."
41 ls -l /usr/bin > bin.file
42 ls -l bin.file    #  Cerca di elencare il contenuto del file che si
43                   #+ è tentato di creare.
44
45 echo
46
47 exit 0
```

Capitolo 22. Sostituzione di processo

La *sostituzione di processo* è analoga alla sostituzione di comando. La sostituzione di comando imposta una variabile al risultato di un comando, come **elenco_dir=`ls -al`** o **xref=$(grep parola filedati)**. La sostituzione di processo, invece, invia l'output di un processo ad un altro processo (in altre parole, manda il risultato di un comando ad un altro comando).

Struttura della sostituzione di processo

comando tra parentesi

> **>(comando)**
>
> **<(comando)**
>
> Queste istanze danno inizio alla sostituzione di processo. Per inviare i risultati del processo tra parentesi ad un altro processo, vengono usati i file `/dev/fd/<n>`. [1]

Non vi è nessuno spazio tra "<" o ">" e le parentesi. Se ce ne fosse uno verrebbe visualizzato un messaggio d'errore.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63
```

Bash crea una pipe con due descrittori di file, `--fIn` e `fOut--`. Lo `stdin` di true si connette a `fOut` (dup2(fOut, 0)), quindi Bash passa `/dev/fd/fIn` come argomento ad **echo**. Sui sistemi che non dispongono dei file `/dev/fd/<n>`, Bash può usare dei file temporanei. (Grazie, S.C.)

Con la sostituzione di processo si possono confrontare gli output di due diversi comandi, o anche l'output di differenti opzioni dello stesso comando.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
        total 20
        drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
        drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
        -rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
```

```
        -rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
        -rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
```

Utilizzare la sostituzione di processo per confrontare il contenuto di due directory (per verificare quali file sono presenti nell'una, ma non nell'altra):

```
  1 diff <(ls $prima_directory) <(ls $seconda_directory)
```

Alcuni altri usi ed impieghi della sostituzione di processo

```
  1 cat <(ls -l)
  2 # Uguale a     ls -l | cat
  3
  4 sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
  5 # Elenca tutti i file delle 3 directory principali 'bin' e li ordina.
  6 # Notate che a 'sort' vengono inviati tre (contateli) distinti comandi.
  7
  8
  9 diff <(comando1) <(comando2)  # Fornisce come output le differenze dei
comandi.
 10
 11
 12 tar cf >(bzip2 -c > file.tar.bz2) $nome_directory
 13 # Richiama "tar cf /dev/fd/?? $nome_directory" e "bzip2 -c > file.tar.bz2".
 14 #
 15 # A causa della funzionalità di sistema di /dev/fd/<n>,
 16 # non occorre che la pipe tra i due comandi sia una named pipe.
 17 #
 18 # Questa può essere emulata.
 19 #
 20 bzip2 -c < pipe > file.tar.bz2&
 21 tar cf pipe $nome_directory
 22 rm pipe
 23 #        oppure
 24 exec 3>&1
 25 tar cf /dev/fd/4 $nome_directory 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2
3>&-
 26 exec 3>&-
 27
 28
 29 # Grazie Stepane Chazelas.
```

Un lettore ci ha inviato il seguente, interessante esempio di sostituzione di processo.

```
  1 # Frammento di script preso dalla distribuzione SuSE:
  2
  3 while read  des what mask iface; do
  4 # Alcuni comandi ...
  5 done < <(route -n)
  6
  7
  8 # Per verificarlo, facciamogli fare qualcosa.
  9 while read  des what mask iface; do
 10   echo $des $what $mask $iface
 11 done < <(route -n)
 12
 13 # Output:
 14 # Kernel IP routing table
```

```
 15 # Destination Gateway Genmask Flags Metric Ref Use Iface
 16 # 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
 17
 18
 19
 20 # Come ha puntualizzato Stephane Chazelas, una forma analoga, più facile da
comprendere, è:
 21 route -n |
 22   while read des what mask iface; do  # Le variabili vengono impostate
 23                                       #+ con l'output della pipe.
 24     echo $des $what $mask $iface
 25   done  #  Produce lo stesso output del precedente.
 26         #  Tuttavia, come rileva Ulrich Gayer . . .
 27         #+ questa forma semplificata usa una subshell per il ciclo while
 28         #+ e, quindi, le variabili scompaiono quando la pipe termina.
 29
 30
 31
 32 #  Filip Moritz fa notare, comunque, che esiste una sottile differenza
 33 #+ tra i due esempi precedenti, come viene mostrato di seguito.
 34
 35 (
 36 route -n | while read x; do ((y++)); done
 37 echo $y # $y risulta ancora non impostata
 38
 39 while read x; do ((y++)); done < <(route -n)
 40 echo $y # $y contiene il numero delle righe dell'output di route -n
 41 )
 42
 43 Più in generale
 44 (
 45 : | x=x
 46 # sembra dare inizio ad una subshell come
 47 : | ( x=x )
 48 # mentre
 49 x=x < <(:)
 50 # no
 51 )
 52
 53 # È utile per la verifica di csv* e operazioni analoghe.
 54 # Ed è quello che, in effetti, fa il frammento di codice originale SuSE.
 55 #* Comma Separated Values - Valori separati da virgole [N.d.T.]
```

Note

[1] Ha lo stesso effetto di una named pipe (file temporaneo) che, infatti, una volta erano usate nella sostituzione di processo.

Capitolo 23. Funzioni

Sommario

Come i "veri" linguaggi di programmazione, anche Bash dispone delle funzioni, sebbene in un'implementazione un po' limitata. Una funzione è una subroutine, un blocco di codice che rende

disponibile una serie di operazioni, una "scatola nera" che esegue un compito specifico. Ogni qual volta vi è del codice che si ripete o quando un compito viene iterato con leggere variazioni, allora è il momento di prendere in considerazione l'uso di una funzione.

function *nome_funzione* {
comando...
}

oppure

nome_funzione () {
comando...
}

Questa seconda forma è quella che rallegra i cuori dei programmatori C (ed è più portabile).

Come nel C, la parentesi graffa aperta può, opzionalmente, comparire nella riga successiva a quella del nome della funzione.

nome_funzione ()
{
comando...
}

Le funzioni vengono richiamate, *messe in esecuzione*, semplicemente invocando i loro nomi.

Esempio 23-1. Semplici funzioni

```
 1 #!/bin/bash
 2
 3 SOLO_UN_SECONDO=1
 4
 5 strana ()
 6 { # Questo a proposito della semplicità delle funzioni.
 7   echo "Questa è una funzione strana."
 8   echo "Ora usciamo dalla funzione strana."
 9 } # La dichiarazione della funzione deve precedere la sua chiamata.
10
11
12 divertimento ()
13 { # Una funzione un po' più complessa.
14   i=0
15   RIPETIZIONI=30
16
17   echo
18   echo "Ed ora, che il divertimento abbia inizio."
19   echo
20
21   sleep $SOLO_UN_SECONDO    # Hey, aspetta un secondo!
22   while [ $i -lt $RIPETIZIONI ]
23   do
24     echo "--------LE FUNZIONI--------->"
25     echo "<----------SONO--------------"
26     echo "<--------DIVERTENTI--------->"
27     echo
28     let "i+=1"
```

```
 29   done
 30 }
 31
 32   # Ora, richiamiamo le funzioni.
 33
 34 strana
 35 divertimento
 36
 37 exit 0
```

La definizione della funzione deve precedere la sua prima chiamata. Non esiste alcun metodo per "dichiarare" la funzione, come, ad esempio, nel C.

```
  1 f1
  2 # Dà un messaggio d'errore poiché la funzione "f1" non è stata ancora
definita.
  3
  4 declare -f f1      # Neanche questo aiuta.
  5 f1                 # Ancora un messaggio d'errore.
  6
  7 # Tuttavia...
  8
  9
 10 f1 ()
 11 {
 12   echo "Chiamata della funzione \"f2\" dalla funzione \"f1\"."
 13   f2
 14 }
 15
 16 f2 ()
 17 {
 18   echo "Funzione \"f2\"."
 19 }
 20
 21 f1  #  La funzione "f2", in realtà, viene chiamata solo a questo punto,
 22     #+ sebbene vi si faccia riferimento prima della sua definizione.
 23     #  Questo è consentito.
 24
 25     # Grazie, S.C.
```

È anche possibile annidare una funzione in un'altra, sebbene non sia molto utile.

```
  1 f1 ()
  2 {
  3
  4   f2 () # annidata
  5   {
  6       echo "Funzione \"f2\", all'interno di \"f1\"."
  7   }
  8
  9 }
 10
 11 f2  # Restituisce un messaggio d'errore.
 12     # Sarebbe inutile anche farla precedere da "declare -f f2".
 13
 14 echo
 15
 16 f1  #  Non fa niente, perché richiamare"f1" non implica richiamare
 17     #+ automaticamente "f2".
 18 f2  #  Ora è tutto a posto, "f2" viene eseguita, perché la sua
```

```
 19     #+ definizione è stata resa visibile tramite la chiamata di "f1".
 20
 21     # Grazie, S.C.
```

Le dichiarazioni di funzione possono comparire in posti impensati, anche dove dovrebbe trovarsi un comando.

```
  1 ls -l | foo() { echo "foo"; }  # Consentito, ma inutile.
  2
  3
  4
  5 if [ "$USER" = bozo ]
  6 then
  7   saluti_bozo ()   # Definizione di funzione inserita in un costrutto
if/then.
  8   {
  9       echo "Ciao, Bozo."
 10   }
 11 fi
 12
 13 saluti_bozo        # Funziona solo per Bozo, agli altri utenti dà un errore.
 14
 15
 16 # Qualcosa di simile potrebbe essere utile in certi contesti.
 17 NO_EXIT=1          # Abilita la definizione di funzione seguente.
 18
 19 [[ $NO_EXIT -eq 1 ]] && exit() { true; }     #  Definizione di funzione
 20                                              #+ in una "lista and".
 21 # Se $NO_EXIT è uguale a 1, viene dichiarata "exit ()".
 22 # Così si disabilita il builtin "exit" rendendolo un alias di "true".
 23
 24 exit  # Viene invocata la funzione "exit ()", non il builtin "exit".
 25
 26 # Grazie, S.C.
```

23.1. Funzioni complesse e complessità delle funzioni

Le funzioni possono elaborare gli argomenti che ad esse vengono passati e restituire un exit status allo script per le successive elaborazioni.

```
  1 nome_funzione $arg1 $arg2
```

La funzione fa riferimento agli argomenti passati in base alla loro posizione (come se fossero parametri posizionali), vale a dire, `$1`, `$2`, eccetera.

Esempio 23-2. Funzione con parametri

```
  1 #!/bin/bash
  2 # Funzioni e parametri
  3
  4 DEFAULT=predefinito               # Valore predefinito del parametro
  5
  6 funz2 () {
  7    if [ -z "$1" ]                 # Il parametro nr.1 è vuoto (lunghezza
```

```
zero)?
  8    then
  9         echo "-Il parametro nr.1 ha lunghezza zero.-"  #  O non è stato
passato
 10                                                        #+ alcun parametro.
 11    else
 12         echo "-Il parametro nr.1 è \"$1\".-"
 13    fi
 14
 15    variabile=${1-$DEFAULT}        #  Cosa rappresenta
 16    echo "variabile = $variabile"  #+ la sostituzione di parametro?
 17                                   #  -----------------------------------
 18                                   #  Fa distinzione tra nessun parametro e
 19                                   #+ parametro nullo.
 20
 21    if [ "$2" ]
 22    then
 23         echo "-Il parametro nr.2 è \"$2\".-"
 24    fi
 25
 26    return 0
 27 }
 28
 29 echo
 30
 31 echo "Non viene passato niente."
 32 funz2                          # Richiamata senza alcun parametro
 33 echo
 34
 35
 36 echo "Viene passato un parametro vuoto."
 37 funz2 ""                      # Richiamata con un parametro di lunghezza
zero
 38 echo
 39
 40 echo "Viene passato un parametro nullo."
 41 funz2 "$param_non_inizializ"   # Richiamata con un parametro non
inizializzato
 42 echo
 43
 44 echo "Viene passato un parametro."
 45 funz2 primo           # Richiamata con un parametro
 46 echo
 47
 48 echo "Vengono passati due parametri."
 49 funz2 primo secondo   # Richiamata con due parametri
 50 echo
 51
 52 echo "Vengono passati \"\" \"secondo\"."
 53 funz2 "" secondo      # Richiamata con il primo parametro di lunghezza zero
 54 echo                  # e una stringa ASCII come secondo.
 55
 56 exit 0
```

Il comando shift opera sugli argomenti passati alle funzioni (vedi Esempio 34-12).

Rispetto ad alcuni altri linguaggi di programmazione, gli script di shell normalmente passano i parametri alle funzioni solo per valore. I nomi delle variabili (che in realtà sono dei puntatori), se passati come parametri alle funzioni, vengono trattati come stringhe. *Le funzioni interpretano i loro argomenti letteralmente*.

La referenziazione indiretta a variabili (vedi Esempio 35-2) offre una specie di meccanismo, un po' goffo, per passare i puntatori a variabile alle funzioni.

Esempio 23-3. Passare una referenziazione indiretta a una funzione

```
 1 #!/bin/bash
 2 # ind-func.sh: Passare una referenziazione indiretta a una funzione.
 3
 4 var_echo ()
 5 {
 6 echo "$1"
 7 }
 8
 9 messaggio=Ciao
10 Ciao=Arrivederci
11
12 var_echo "$messaggio"      # Ciao
13 # Adesso passiamo una referenziazione indiretta alla funzione.
14 var_echo "${!messaggio}"   # Arrivederci
15
16 echo "-------------"
17
18 # Cosa succede se modifichiamo il contenuto della variabile "Ciao"?
19 Ciao="Ancora ciao!"
20 var_echo "$messaggio"      # Ciao
21 var_echo "${!messaggio}"   # Ancora ciao!
22
23 exit 0
```

La domanda logica successiva è se i parametri possono essere dereferenziati *dopo* essere stati passati alla funzione.

Esempio 23-4. Dereferenziare un parametro passato a una funzione

```
 1 #!/bin/bash
 2 # dereference.sh
 3 # Dereferenziare un parametro passato ad una funzione.
 4 # Script di Bruce W. Clare.
 5
 6 dereferenzia ()
 7 {
 8      y=\$"$1"   # Nome della variabile.
 9      echo $y    # $Prova
10
11      x=`eval "expr \"$y\" "`
12      echo $1=$x
13      eval "$1=\"Un testo diverso \""  # Assegna un nuovo valore.
14 }
15
16 Prova="Un testo"
17 echo $Prova "prima"    # Un testo prima
18
19 dereferenzia Prova
20 echo $Prova "dopo"     # Un testo diverso dopo
21
22 exit 0
```

Esempio 23-5. Ancora, dereferenziare un parametro passato a una funzione

```
  1 #!/bin/bash
  2 # ref-params.sh: Dereferenziare un parametro passato a una funzione.
  3 #                (Esempio complesso)
  4
  5 ITERAZIONI=3  # Numero di input da immettere.
  6 contai=1
  7
  8 lettura () {
  9   #  Richiamata nella forma lettura nomevariabile,
 10   #+ visualizza il dato precedente tra parentesi quadre come dato
predefinito,
 11   #+ quindi chiede un nuovo valore.
 12
 13   local var_locale
 14
 15   echo -n "Inserisci un dato "
 16   eval 'echo -n "[$'$1'] "'  #  Dato precedente.
 17 # eval echo -n "[\$$1] "     #  Più facile da capire,
 18                              #+ ma si perde lo spazio finale al prompt.
 19   read var_locale
 20   [ -n "$var_locale" ] && eval $1=\$var_locale
 21
 22   #  "Lista And": se "var_locale" è presente allora viene impostata
 23   #+ al valore di "$1".
 24 }
 25
 26 echo
 27
 28 while [ "$contai" -le "$ITERAZIONI" ]
 29 do
 30   lettura var
 31   echo "Inserimento nr.$contai = $var"
 32   let "contai += 1"
 33   echo
 34 done
 35
 36
 37 # Grazie a Stephane Chazelas per aver fornito questo istruttivo esempio.
 38
 39 exit 0
```

Exit e Return

exit status

Le funzioni restituiscono un valore, chiamato *exit status*. L'exit status può essere specificato in maniera esplicita con un'istruzione **return**, altrimenti corrisponde all'exit status dell'ultimo comando della funzione (0 in caso di successo, un codice d'errore diverso da zero in caso contrario). Questo exit status può essere usato nello script facendovi riferimento tramite $?. Questo meccanismo consente alle funzioni di avere un "valore di ritorno" simile a quello delle funzioni del C.

return

Termina una funzione. Il comando **return** [1] può avere opzionalmente come argomento un *intero*, che viene restituito allo script chiamante come "exit status" della funzione. Questo exit status viene assegnato alla variabile $?.

Esempio 23-6. Il maggiore di due numeri

```
  1 #!/bin/bash
  2 # max.sh: Maggiore di due numeri.
  3
  4 E_ERR_PARAM=-198    # Se vengono passati meno di 2 parametri alla
funzione.
  5 UGUALI=-199         # Valore di ritorno se i due numeri sono uguali.
  6
  7 max2 ()             # Restituisce il maggiore di due numeri.
  8 {                   # Nota: i numeri confrontati devono essere
minori di 257.
  9 if [ -z "$2" ]
 10 then
 11   return $E_ERR_PARAM
 12 fi
 13
 14 if [ "$1" -eq "$2" ]
 15 then
 16   return $UGUALI
 17 else
 18   if [ "$1" -gt "$2" ]
 19   then
 20     return $1
 21   else
 22     return $2
 23   fi
 24 fi
 25 }
 26
 27 max2 33 34
 28 val_ritorno=$?
 29
 30 if [ "$val_ritorno" -eq $E_ERR_PARAM ]
 31 then
 32   echo "Bisogna passare due parametri alla funzione."
 33 elif [ "$val_ritorno" -eq $UGUALI ]
 34   then
 35     echo "I due numeri sono uguali."
 36 else
 37     echo "Il maggiore dei due numeri è $val_ritorno."
 38 fi
 39
 40
 41 exit 0
 42
 43 #  Esercizio (facile):
 44 #  ------------------
 45 #  Trasformatelo in uno script interattivo,
 46 #+ cioè, deve essere lo script a richiedere l'input (i due numeri).
```


Per fare in modo che una funzione possa restituire una stringa o un array , si deve fare ricorso ad una variabile dedicata.

```
  1 conteggio_righe_di_etc_passwd()
  2 {
  3   [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l <
/etc/passwd))
  4   #  Se /etc/passwd ha i permessi di lettura, imposta
REPLY al
  5   #+ numero delle righe.
```

```
  6   #  Restituisce o il valore del parametro o
un'informazione di stato.
  7   # 'echo' sembrerebbe non necessario, ma . . .
  8   #+ rimuove dall'output gli spazi in eccesso.
  9 }
 10
 11 if conteggio_righe_di_etc_passwd
 12 then
 13   echo "Ci sono $REPLY righe in /etc/passwd."
 14 else
 15   echo "Non posso contare le righe in /etc/passwd."
 16 fi
 17
 18 # Grazie, S.C.
```

Esempio 23-7. Convertire i numeri arabi in numeri romani

```
  1 #!/bin/bash
  2
  3 # Conversione di numeri arabi in numeri romani
  4 # Intervallo: 0 - 200
  5 # È rudimentale, ma funziona.
  6
  7 #  Viene lasciato come esercizio l'estensione dell'intervallo e
  8 #+ altri miglioramenti dello script.
  9
 10 # Utilizzo: numero da convertire in numero romano
 11
 12 LIMITE=200
 13 E_ERR_ARG=65
 14 E_FUORI_INTERVALLO=66
 15
 16 if [ -z "$1" ]
 17 then
 18   echo "Utilizzo: `basename $0` numero-da-convertire"
 19   exit $E_ERR_ARG
 20 fi
 21
 22 num=$1
 23 if [ "$num" -gt $LIMITE ]
 24 then
 25   echo "Fuori intervallo!"
 26   exit $E_FUORI_INTERVALLO
 27 fi
 28
 29 calcola_romano ()   # Si deve dichiarare la funzione prima di
richiamarla.
 30 {
 31 numero=$1
 32 fattore=$2
 33 rchar=$3
 34 let "resto = numero - fattore"
 35 while [ "$resto" -ge 0 ]
 36 do
 37   echo -n $rchar
 38   let "numero -= fattore"
 39   let "resto = numero - fattore"
 40 done
 41
 42 return $numero
 43        # Esercizio:
```

```
 44        # ----------
 45        # Spiegate come opera la funzione.
 46        # Suggerimento: divisione per mezzo di sottrazioni
successive.
 47 }
 48
 49
 50 calcola_romano $num 100 C
 51 num=$?
 52 calcola_romano $num 90 LXXXX
 53 num=$?
 54 calcola_romano $num 50 L
 55 num=$?
 56 calcola_romano $num 40 XL
 57 num=$?
 58 calcola_romano $num 10 X
 59 num=$?
 60 calcola_romano $num 9 IX
 61 num=$?
 62 calcola_romano $num 5 V
 63 num=$?
 64 calcola_romano $num 4 IV
 65 num=$?
 66 calcola_romano $num 1 I
 67
 68 echo
 69
 70 exit 0
```

Vedi anche Esempio 10-28.

Il più grande intero positivo che una funzione può restituire è 255. Il comando **return** è strettamente legato al concetto di exit status, e ciò è la causa di questa particolare limitazione. Fortunatamente, esistono diversi espedienti per quelle situazioni che richiedono, come valore di ritorno della funzione, un intero maggiore di 255.

Esempio 23-8. Verificare valori di ritorno di grandi dimensioni in una funzione

```
  1 #!/bin/bash
  2 # return-test.sh
  3
  4 # Il maggiore valore positivo che una funzione può
restituire è 255.
  5
  6 val_ritorno ()        # Restituisce tutto quello che gli
viene passato.
  7 {
  8   return $1
  9 }
 10
 11 val_ritorno 27         # o.k.
 12 echo $?               # Restituisce 27.
 13
 14 val_ritorno 255        # Ancora o.k.
 15 echo $?               # Restituisce 255.
 16
 17 val_ritorno 257        # Errore!
 18 echo $?               # Restituisce 1 (codice d'errore
generico).
```

```
 19
 20 #
============================================================
 21 val_ritorno -151896    # Funziona con grandi numeri
negativi?
 22 echo $?               # Restituirà -151896?
 23                       # No! Viene restituito 168.
 24 #  Le versioni di Bash precedenti alla 2.05b permettevano
 25 #+ valori di ritorno di grandi numeri negativi.
 26 #  Quelle più recenti non consentono questa scappatoia.
 27 #  Ciò potrebbe rendere malfunzionanti i vecchi script.
 28 #  Attenzione!
 29 #
============================================================
 30
 31 exit 0
```

Un espediente per ottenere un intero di grandi dimensioni consiste semplicemente nell'assegnare il "valore di ritorno" ad una variabile globale.

```
  1 Val_Ritorno=  #  Variabile globale che contiene un valore
di ritorno
  2               #+ della funzione maggiore di 255.
  3
  4 ver_alt_ritorno ()
  5 {
  6   fvar=$1
  7   Val_Ritorno=$fvar
  8   return      # Restituisce 0 (successo).
  9 }
 10
 11 ver_alt_ritorno 1
 12 echo $?                                    # 0
 13 echo "valore di ritorno = $Val_Ritorno"    # 1
 14
 15 ver_alt_ritorno 256
 16 echo "valore di ritorno = $Val_Ritorno"    # 256
 17
 18 ver_alt_ritorno 257
 19 echo "valore di ritorno = $Val_Ritorno"    # 257
 20
 21 ver_alt_ritorno 25701
 22 echo "valore di ritorno = $Val_Ritorno"    # 25701
```

Un metodo anche più elegante consiste semplicemente nel visualizzare allo `stdout` il "valore di ritorno" della funzione con il comando **echo** e poi "catturarlo" per mezzo della sostituzione di parametro. Per una discussione sull'argomento vedi la Sezione 34.7.

Esempio 23-9. Confronto di due interi di grandi dimensioni

```
  1 #!/bin/bash
  2 # max2.sh: Maggiore di due GRANDI interi.
  3
  4 #  È il precedente esempio "max.sh" ,
  5 #+ modificato per consentire il confronto di grandi numeri.
  6
  7 UGUALI=0            # Valore di ritorno se i due parametri sono
uguali.
```

```
  8 E_ERR_PARAM=-99999  # Numero di parametri passati alla funzione
insufficiente.
  9
 10 max2 ()             # "Restituisce" il maggiore di due numeri.
 11 {
 12 if [ -z "$2" ]
 13 then
 14   echo $E_ERR_PARAM
 15   return
 16 fi
 17
 18 if [ "$1" -eq "$2" ]
 19 then
 20   echo $UGUALI
 21   return
 22 else
 23   if [ "$1" -gt "$2" ]
 24   then
 25       valritorno=$1
 26   else
 27       valritorno=$2
 28   fi
 29 fi
 30
 31 echo $valritorno    # Visualizza (allo stdout) il valore invece di
restituirlo.
 32
 33 }
 34
 35
 36 val_ritorno=$(max2 33001 33997)
 37 #  Si tratta, in realtà, di una forma di sostituzione di comando:
 38 #+ che tratta una funzione come se fosse un comando
 39 #+ e che assegna lo stdout della funzione alla variabile
'val_ritorno' . . .
 40
 41
 42 # ============================== RISULTATO
==============================
 43 if [ "$val_ritorno" -eq "$E_ERR_PARAM" ]
 44 then
 45   echo "Errore nel numero di parametri passati alla funzione di
confronto!"
 46 elif [ "$val_ritorno" -eq "$UGUALI" ]
 47   then
 48       echo "I due numeri sono uguali."
 49   else
 50       echo "Il maggiore dei due numeri è $val_ritorno."
 51 fi
 52 #
================================================================================
==
 53
 54 exit 0
 55
 56 #  Esercizi:
 57 #  --------
 58 #  1) Trovate un modo più elegante per verificare
 59 #+    il numero di parametri passati alla funzione.
 60 #  2) Semplificate la struttura if/then presente nel blocco
"RISULTATO."
 61 #  3) Riscrivete lo script in modo che l'input sia dato dai
```

```
parametri passati
 62 #+    da riga di comando.
```

Vedi anche Esempio A-8.

Esercizio: Utilizzando le conoscenze fin qui acquisite, si estenda il precedente esempio dei numeri romani in modo che accetti un input arbitrario maggiore di 255.

Redirezione

Redirigere lo stdin di una funzione

Una funzione è essenzialmente un blocco di codice, il che significa che il suo stdin può essere rediretto (come in Esempio 3-1).

Esempio 23-10. Il vero nome dal nome utente

```
  1 #!/bin/bash
  2 # realname.sh
  3
  4 # Partendo dal nome dell'utente, ricava il "vero nome" da
/etc/passwd.
  5
  6
  7 CONTOARG=1  # Si aspetta un argomento.
  8 E_ERR_ARG=65
  9
 10 file=/etc/passwd
 11 modello=$1
 12
 13 if [ $# -ne "$CONTOARG" ]
 14 then
 15   echo "Utilizzo: `basename $0` NOME-UTENTE"
 16   exit $E_ERR_ARG
 17 fi
 18
 19 ricerca ()      #  Esamina il file alla ricerca del modello, quindi
visualizza
 20                 #+ la parte rilevante della riga.
 21 {
 22 while read riga # "while" non necessariamente vuole la "[
condizione]"
 23 do
 24   echo "$riga" | grep $1 | awk -F":" '{ print $5 }' #  awk deve
usare
 25                                                     #+ i ":" come
delimitatore.
 26 done
 27 } <$file  # Redirige nello stdin della funzione.
 28
 29 ricerca $modello
 30
 31 # Certo, l'intero script si sarebbe potuto ridurre a
 32 #       grep MODELLO /etc/passwd | awk -F":" '{ print $5 }'
 33 # oppure
 34 #       awk -F: '/MODELLO/ {print $5}'
 35 # oppure
 36 #       awk -F: '($1 == "nomeutente") { print $5 }' #  il vero nome
```

```
dal
 37                                                    #+ nome utente
 38 # Tuttavia, non sarebbe stato altrettanto istruttivo.
 39
 40 exit 0
```

Esiste un'alternativa, un metodo che confonde forse meno, per redirigere lo `stdin` di una funzione. Questo comporta la redirezione dello `stdin` in un blocco di codice compreso tra parentesi graffe all'interno della funzione.

```
  1 # Invece di:
  2 Funzione ()
  3 {
  4  ...
  5 } < file
  6
  7 # Provate:
  8 Funzione ()
  9 {
 10   {
 11     ...
 12    } < file
 13 }
 14
 15 # Analogamente,
 16
 17 Funzione ()  # Questa funziona.
 18 {
 19   {
 20   echo $*
 21   } | tr a b
 22 }
 23
 24 Funzione ()  # Questa, invece, no.
 25 {
 26   echo $*
 27 } | tr a b   # In questo caso è obbligatorio il blocco di codice
annidato.
 28
 29
 30 # Grazie, S.C.
```

Note

[1] Il comando **return** è un builtin Bash.

23.2. Variabili locali

Cosa rende una variabile "locale"?

variabili locali

Una variabile dichiarata come *local* è quella che è visibile solo all'interno del blocco di codice in cui appare. Ha "ambito" locale. In una funzione una *variabile locale* ha significato solo all'interno del blocco di codice della funzione.

Esempio 23-11. Visibilità di una variabile locale

```
  1 #!/bin/bash
  2 # Variabili globali e locali in una funzione.
  3
  4 funz ()
  5 {
  6   local var_locale=23       # Dichiarata come variabile locale.
  7   echo                      # Utilizza il builtin 'local'.
  8   echo "\"var_locale\" nella funzione = $var_locale"
  9   var_globale=999           # Non dichiarata come locale.
 10                             # Viene impostata per default come
globale.
 11   echo "\"var_globale\" nella funzione = $var_globale"
 12 }
 13
 14 funz
 15
 16 #  Ora, per controllare se la variabile locale "var_locale" esiste
al di fuori
 17 #+ della funzione.
 18
 19 echo
 20 echo "\"var_locale\" al di fuori della funzione = $var_locale"
 21                                # $var_locale al di fuori della
funzione =
 22                                # No, $var_locale non ha visibilità
globale.
 23 echo "\"var_globale\" al di fuori della funzione = $var_globale"
 24                                # $var_globale al di fuori della
funzione = 999
 25                                # $var_globale è visibile globalmente
 26 echo
 27
 28 exit 0
 29 #  A differenza del C, una variabile Bash dichiarata all'interno di
una funzione
 30 #+ è locale "solo" se viene dichiarata come tale.
```

Prima che una funzione venga richiamata, *tutte* le variabili dichiarate all'interno della funzione sono invisibili al di fuori del corpo della funzione stessa, non soltanto quelle esplicitamente dichiarate come *locali*.

```
  1 #!/bin/bash
  2
  3 funz ()
  4 {
  5 var_globale=37    #  Visibile solo all'interno del blocco
della funzione
  6                   #+ prima che la stessa venga
richiamata.
  7 }                 #  FINE DELLA FUNZIONE
  8
  9 echo "var_globale = $var_globale"  #  var_globale =
 10                                    #  La funzione "funz"
non è ancora stata
```

```
 11                                      #+ chiamata, quindi
$var_globale qui non è
 12                                      #+ visibile.
 13
 14 funz
 15 echo "var_globale = $var_globale"  # var_globale = 37
 16                                      # È stata impostata
richiamando la funzione.
```

23.2.1. Le variabili locali aiutano a realizzare la ricorsività.

Le variabili locali consentono la ricorsività, [1] ma questa pratica implica, generalmente, un carico computazionale elevato e, in definitiva, *non* viene raccomandata in uno script di shell. [2]

Esempio 23-12. Ricorsività tramite una variabile locale

```
 1 #!/bin/bash
 2
 3 #               fattoriale
 4 #               ----------
 5
 6
 7 # Bash permette la ricorsività?
 8 # Ebbene, sì, ma...
 9 # Dovreste avere dei sassi al posto del cervello per usarla.
10
11
12 MAX_ARG=5
13 E_ERR_ARG=65
14 E_ERR_MAXARG=66
15
16
17 if [ -z "$1" ]
18 then
19   echo "Utilizzo: `basename $0` numero"
20   exit $E_ERR_ARG
21 fi
22
23 if [ "$1" -gt $MAX_ARG ]
24 then
25   echo "Valore troppo grande (il massimo è 5)."
26   #  Torniamo alla realtà.
27   #  Se fosse necessario un numero maggiore di questo,
28   #+ riscrivete lo script in un vero linguaggio di programmazione.
29   exit $E_ERR_MAXARG
30 fi
31
32 fatt ()
33 {
34   local numero=$1
35   #  La variabile "numero" deve essere dichiarata locale,
36   #+ altrimenti questa funzione non svolge il suo compito.
37   if [ "$numero" -eq 0 ]
38   then
39       fattoriale=1    # Il fattoriale di 0 è 1.
40   else
41       let "decrnum = numero - 1"
42       fatt $decrnum  # Chiamata ricorsiva della funzione.
43       let "fattoriale = $numero * $?"
44    fi
```

```
45
46    return $fattoriale
47 }
48
49 fatt $1
50 echo "Il fattoriale di $1 è $?."
51
52 exit 0
```

Vedi anche Esempio A-17 per una dimostrazione di ricorsività in uno script. Si faccia attenzione che la ricorsività sfrutta intensivamente le risorse, viene eseguita lentamente e, di conseguenza, il suo uso, in uno script, non è appropriato.

Note

[1] Herbert Mayer definisce la *ricorsività* come "...esprimere un algoritmo usando una versione semplificata di quello stesso algoritmo...". Una funzione ricorsiva è quella che richiama sé stessa.

[2] Troppi livelli di ricorsività possono mandare in crash lo script con un messaggio di segmentation fault.

```
  1 #!/bin/bash
  2
  3 #  Attenzione: è probabile che l'esecuzione di questo script blocchi il
sistema!
  4 #  Se siete fortunati, verrete avvertiti da un segmentation fault prima
che
  5 #+ tutta la memoria disponibile venga occupata.
  6
  7 funzione_ricorsiva ()
  8 {
  9 (( $1 < $2 )) && f $(( $1 + 1 )) $2
 10 #  Finché il primo parametro è inferiore al secondo,
 11 #+ il primo viene incrementato ed il tutto si ripete.
 12 }
 13
 14 funzione_ricorsiva 1 50000  # Ricorsività di 50,000 livelli!
 15 #  Molto probabilmente segmentation fault (in base alla dimensione
dello stack,
 16 #+ impostato con ulimit -m).
 17
 18 #  Una ricorsività così elevata potrebbe causare un segmentation fault
 19 #+ anche in un programma in C, a seguito dell'uso di tutta la memoria
 20 #+ allocata nello stack.
 21
 22
 23 echo "Probabilmente questo messaggio non verrà visualizzato."
 24 exit 0  # Questo script non terminarà normalmente.
 25
 26 #  Grazie, Stephane Chazelas.
```

23.3. Ricorsività senza variabili locali

Una funzione può richiamare se stessa ricorsivamente anche senza l'impiego di variabili locali.

Esempio 23-13. La torre di Hanoi

```
 1 #! /bin/bash
 2 #
 3 # La Torre di Hanoi
 4 # Script Bash
 5 # Copyright (C) 2000 Amit Singh. Tutti i diritti riservati.
 6 # http://hanoi.kernelthread.com
 7 #
 8 # Ultima verifica eseguita con la versione bash 2.05b.0(13)-release
 9 #
10 #  Usato in "Advanced Bash Scripting Guide"
11 #+ con il permesso dell'autore dello script.
12 #  Commentato e leggermente modificato dall'autore di ABS.
13
14 #==========================================================================#
15 #  La Torre di Hanoi è un vecchio rompicapo matematico.
16 #  Ci sono tre pioli verticali inseriti in una base.
17 #  Nel primo piolo è impilata una serie di anelli rotondi.
18 #  Gli anelli sono dei dischi piatti con un foro al centro,
19 #+ in modo che possano essere infilati nei pioli.
20 #  I dischi hanno diametri diversi e sono impilati in ordine
21 #+ decrescente in base alla loro dimensione.
22 #  Quello più piccolo si trova nella posizione più alta,
23 #+ quello più grande alla base.
24 #
25 #  Lo scopo è quello di trasferire la pila di dischi
26 #+ in uno degli altri pioli.
27 #  Si può spostare solo un disco alla volta.
28 #  È consentito rimettere i dischi nel piolo iniziale.
29 #  È permesso mettere un disco su un altro di dimensione maggiore,
30 #+ ma *non* viceversa.
31 #  Ancora, è proibito collocare un disco su uno di minor diametro.
32 #
33 #  Con un numero ridotto di dischi, sono necessari solo pochi spostamenti.
34 #+ Per ogni disco aggiuntivo,
35 #+ il numero degli spostamenti richiesti approssimativamente raddoppia
36 #+ e la "strategia" diventa sempre più complessa.
37 #
38 #  Per ulteriori informazioni, vedi http://hanoi.kernelthread.com.
39 #
40 #
41 #         ...                    ...                    ...
42 #         | |                    | |                    | |
43 #        _|_|_                   | |                    | |
44 #       |_____|                  | |                    | |
45 #      |_______|                 | |                    | |
46 #     |_________|                | |                    | |
47 #    |___________|               | |                    | |
48 #   |             |              | |                    | |
49 # .-------------------------------------------------------------.
50 # |*************************************************************|
51 #          #1                    #2                     #3
52 #
53 #==========================================================================#
54
55
56 E_NOPARAM=66    # Nessun parametro passato allo script.
57 E_ERR_PARAM=67  # Il numero di dischi passato allo script non è valido.
58 Mosse=          # Variabile globale contenente il numero degli spostamenti.
59                 # Modifiche allo script originale.
60
```

```
 61 eseguehanoi() { # Funzione ricorsiva.
 62     case $1 in
 63     0)
 64         ;;
 65     *)
 66         eseguehanoi "$(($1-1))" $2 $4 $3
 67         echo sposto $2 "-->" $3
 68     let "Mosse += 1"  # Modifica allo script originale.
 69         eseguehanoi "$(($1-1))" $4 $3 $2
 70         ;;
 71     esac
 72 }
 73
 74 case $# in
 75 1)
 76     case $(($1>0)) in     # Deve esserci almeno un disco.
 77     1)
 78         eseguehanoi $1 1 3 2
 79         echo "Totale spostamenti = $Mosse"
 80         exit 0;
 81         ;;
 82     *)
 83         echo "$0: numero di dischi non consentito";
 84         exit $E_ERR_PARAM;
 85         ;;
 86     esac
 87     ;;
 88 *)
 89     echo "utilizzo: $0 N"
 90     echo "       Dove \"N\" è il numero dei dischi."
 91     exit $E_NOPARAM;
 92     ;;
 93 esac
 94
 95 # Esercizi:
 96 # ---------
 97 # 1) Eventuali comandi posti in questo punto verrebbero eseguiti?
 98 #    Perché no? (Facile)
 99 # 2) Spiegate il funzionamento della funzione "eseguehanoi".
100 #    (Difficile)
```

Capitolo 24. Alias

Un *alias* Bash, essenzialmente, non è niente più che una scorciatoia di tastiera, un'abbreviazione, un mezzo per evitare di digitare una lunga sequenza di comandi. Se, per esempio, si inserisce la riga **alias lm="ls -l | more"** nel file ~/.bashrc, ogni volta che verrà digitato **lm** da riga di comando, questo sarà automaticamente sostituito con **ls -l | more**. In questo modo si possono evitare lunghe digitazioni da riga di comando nonché dover ricordare combinazioni complesse di comandi ed opzioni. Impostare **alias rm="rm -i"** (modalità di cancellazione interattiva) può evitare moltissimi danni, perché impedisce di perdere inavvertitamente file importanti.

In uno script, gli alias hanno utilità molto limitata. Sarebbe alquanto bello se gli alias potessero assumere alcune delle funzionalità del preprocessore del C, come l'espansione di macro, ma sfortunatamente Bash non espande gli argomenti presenti nel corpo dell'alias. [1] Inoltre, uno script non è in grado di espandere l'alias stesso nei "costrutti composti", come gli enunciati if/then, i cicli e le funzioni. Un'ulteriore limitazione è rappresentata dal fatto che un alias non si espande

ricorsivamente. Quasi invariabilmente, tutto quello che ci piacerebbe fosse fatto da un alias, può essere fatto molto più efficacemente con una funzione.

Esempio 24-1. Alias in uno script

```
  1 #!/bin/bash
  2 #  Invocatelo con un parametro da riga di comando per provare l'ultima
sezione
  3 #+ dello script.
  4
  5 shopt -s expand_aliases
  6 #  È necessario impostare questa opzione, altrimenti lo script non espande
  7 #+ gli alias.
  8
  9
 10 # Innazitutto, divertiamoci un po'.
 11 alias Jesse_James='echo "\"Alias Jesse James\" era una commedia del 1959\
 12 interpretata da Bob Hope."'
 13 Jesse_James
 14
 15 echo; echo; echo;
 16
 17 alias ll="ls -l"
 18 #  Per definire un alias si possono usare sia gli apici singoli (') che
quelli
 19 #+ doppi (").
 20
 21 echo "Prova dell'alias \"ll\":"
 22 ll /usr/X11R6/bin/mk*   #* L'alias funziona.
 23
 24 echo
 25
 26 directory=/usr/X11R6/bin/
 27 prefisso=mk*  # Vediamo se il carattere jolly causa dei problemi.
 28 echo "Variabili \"directory\" + \"prefisso\" = $directory$prefisso"
 29 echo
 30
 31 alias lll="ls -l $directory$prefisso"
 32
 33 echo "Prova dell'alias \"lll\":"
 34 lll           #  Lungo elenco di tutti i file presenti in /usr/X11R6/bin che
 35              #+ iniziano con mk.
 36 # L'alias gestisce correttamente le variabili concatenate e il carattere
jolly.
 37
 38
 39
 40 TRUE=1
 41
 42 echo
 43
 44 if [ TRUE ]
 45 then
 46   alias rr="ls -l"
 47   echo "Prova dell'alias \"rr\" all'interno di un enunciato if/then:"
 48   rr /usr/X11R6/bin/mk*   #* Messaggio d'errore!
 49   # Gli alias non vengono espansi all'interno di enunciati composti.
 50   echo "Comunque, l'alias precedentemente espanso viene ancora
riconosciuto:"
 51   ll /usr/X11R6/bin/mk*
 52 fi
```

```
53
54 echo
55
56 conto=0
57 while [ $conto -lt 3 ]
58 do
59   alias rrr="ls -l"
60   echo "Prova dell'alias \"rrr\" in un ciclo \"while\":"
61   rrr /usr/X11R6/bin/mk*  #* Anche in questo caso l'alias non viene espanso.
62                           #  alias.sh: line 61: rrr: command not found
63   let conto+=1
64 done
65
66 echo; echo
67
68 alias xyz='cat $0'   # Lo script visualizza sé stesso.
69                      # Notate il quoting forte.
70 xyz
71 #  Questo sembra funzionare,
72 #+ sebbene la documentazione Bash suggerisca il contrario.
73 #
74 #  In ogni caso, come ha evidenziato Steve Jacobson,
75 #+ il parametro "$0" viene espanso immediatamente alla
76 #+ dichiarazione dell'alias.
77
78 exit 0
```

Il comando **unalias** elimina un *alias* precedentemente impostato.

Esempio 24-2. unalias: abilitare e disabilitare un alias

```
 1 #!/bin/bash
 2
 3 shopt -s expand_aliases  # Abilita l'espansione degli alias.
 4
 5 alias llm='ls -al | more'
 6 llm
 7
 8 echo
 9
10 unalias llm             # Disabilita l'alias.
11 llm
12 # Dà un messaggio d'errore poiché 'llm' non viene più riconosciuto.
13
14 exit 0
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: command not found
```

Note

[1] Tuttavia, sembra che gli alias possano effettivamente espandere i parametri posizionali.

Capitolo 25. Costrutti lista

I costrutti "lista and" e "lista or" forniscono un mezzo per elaborare consecutivamente un elenco di comandi. Questi possono sostituire efficacemente complessi enunciati **if**/**then** annidati nonché l'enunciato **case**.

Concatenare comandi

lista and

```
1 comando-1 && comando-2 && comando-3 && ... comando-n
```

Ogni comando che a turno deve essere eseguito si accerta che quello precedente abbia restituito come valore di ritorno true (zero). Alla prima restituzione di false (non-zero), la serie dei comandi termina (il primo comando che ha restituito false è l'ultimo che è stato eseguito).

Esempio 25-1. Usare una "lista and" per verificare gli argomenti da riga di comando

```
 1 #!/bin/bash
 2 # "lista and"
 3
 4 if [ ! -z "$1" ] && echo "Argomento nr.1 = $1" && [ ! -z "$2" ] &&\
 5 echo "Argomento nr.2 = $2"
 6
 7 then
 8   echo "Allo script sono stati passati almeno 2 argomenti."
 9   # Tutti i comandi della serie hanno restituito true.
10 else
11   echo "Allo script sono stati passati meno di 2 argomenti."
12   # Almeno uno dei comandi ha restituito false.
13 fi
14 # Notate che "if [ ! -z $1 ]" funziona, ma il suo supposto
equivalente,
15 #  if [ -n $1 ] no. Comunque, l'uso del quoting risolve il problema.
16 #  if [ -n "$1" ] funziona.  State attenti!
17 # In una verifica, è sempre meglio usare le variabili con il
quoting.
18
19
20 # Questo svolge la stesso compito usando solamente enunciati
if/then.
21 if [ ! -z "$1" ]
22 then
23   echo "Argomento nr.1 = $1"
24 fi
25 if [ ! -z "$2" ]
26 then
27   echo "Argomento nr.2 = $2"
28   echo "Allo script sono stati passati almeno 2 argomenti."
29 else
30   echo "Allo script sono stati passati meno di 2 argomenti."
31 fi
32 # È più lungo e meno elegante di una "lista and".
33
34
35 exit 0
```

Esempio 25-2. Un'altra verifica di argomenti da riga di comando utilizzando una "lista and"

```
  1 #!/bin/bash
  2
  3 ARG=1         # Numero degli argomenti attesi.
  4 E_ERR_ARG=65  # Valore d'uscita se il numero di argomenti passati è
errato.
  5
  6 test $# -ne $ARG && echo "Utilizzo: `basename $0` $ARG \
  7 argomento/i" && exit $E_ERR_ARG
  8 #  Se la prima condizione è vera (numero errato di argomenti passati
allo
  9 #+ script), allora vengono eseguiti i comandi successivi e lo script
termina.
 10
 11 # La riga seguente verrà eseguita solo se fallisce la verifica
precedente.
 12 echo "Allo script è stato passato un numero corretto di argomenti."
 13
 14 exit 0
 15
 16 #  Per verificare il valore d'uscita, eseguite "echo $?" dopo che lo
script
 17 #+ è terminato.
```

Naturalmente, una *lista and* può anche essere usata per *impostare* le variabili ad un valore predefinito.

```
  1 arg1=$@       #  Imposta $arg1 al numero di argomenti passati da
riga di
  2               #+ comando, se ce ne sono.
  3
  4 [ -z "$arg1" ] && arg1=DEFAULT
  5               #  Viene impostata a DEFAULT se, da riga di comando,
non è
  6               #+ stato passato niente.
```

lista or

```
  1 comando-1 || comando-2 || comando-3 || ... comando-n
```

Ogni comando che a turno deve essere eseguito si accerta che quello precedente abbia restituito false. Alla prima restituzione di true, la serie dei comandi termina (il primo comando che ha restituito true è l'ultimo che è stato eseguito). Ovviamente è l'inverso della "lista and".

Esempio 25-3. Utilizzare la "lista or" in combinazione con una "lista and"

```
  1 #!/bin/bash
  2
  3 #  delete.sh, utility
di cancellazione di file
non molto intelligente.
  4 #  Utilizzo: delete
nomefile
  5
  6 E_ERR_ARG=65
  7
```

```
  1 # ==> Questi frammenti di codice, presi
dallo script/etc/rc.d/init.d/single
  2 #+==> di Miquel van Smoorenburg,
illustrano l'impiego delle liste "and" e
"or".
  3 # ==> I commenti con la "freccia" sono
stati aggiunti dall'autore del libro.
  4
  5 [ -x /usr/bin/clear ] && /usr/bin/clear
  6   # ==> Se /usr/bin/clear esiste, allora
viene invocato.
  7   # ==> Verificare l'esistenza di un
  8 if [ -z "$1" ]
  9 then
 10   echo "Utilizzo:
`basename $0` nomefile"
 11   exit $E_ERR_ARG  #
Nessun argomento?
Abbandono.
 12 else
 13   file=$1          #
Imposta il nome del file.
 14 fi
 15
 16
 17 [ ! -f "$file" ] &&
echo "File \"$file\" non
trovato. \
 18 Mi rifiuto, in modo
vile, di cancellare un
file inesistente."
 19 #  LISTA AND,
fornisce il messaggio
d'errore se il file non è
presente.
 20 #  Notate il
messaggio di echo che
continua alla riga
successiva per mezzo del
 21 #+ carattere di
escape.
 22
 23 [ ! -f "$file" ] ||
(rm -f $file; echo "File
\"$file\" cancellato.")
 24 # LISTA OR, per
cancellare il file se
presente.
 25
 26 # Notate l'inversione
logica precedente.
 27 # La LISTA AND viene
eseguita se il risultato
è true, la LISTA OR se è
false.
 28
 29 exit 0
```

Se il primo comando di una "lista or" restituisce true, esso verrà eseguito *comunque* .

```
comando prima che venga eseguito
  8   #+==> evita messaggi d'errore e i
conseguenti avvertimenti.
  9
 10   # ==> . . .
 11
 12 # If they want to run something in single
user mode, might as well run it...
 13 for i in /etc/rc1.d/S[0-9][0-9]* ; do
 14         # Check if the script is there.
 15         [ -x "$i" ] || continue
 16   # ==> Se il corrispondente file in $PWD
*non* viene trovato,
 17   #+==> allora "continua" saltando
all'inizio del ciclo.
 18
 19         # Reject backup files and files
generated by rpm.
 20         case "$1" in
 21
*.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
 22                         continue;;
 23         esac
 24         [ "$i" = "/etc/rc1.d/S00single" ]
&& continue
 25   # ==> Imposta il nome dello script, ma
non lo esegue ancora.
 26         $i start
 27 done
 28
 29   # ==> . . .
```

L'exit status di una **lista and** o di una **lista or** corrisponde all'exit status dell'ultimo comando eseguito.

Sono possibili ingegnose combinazioni di liste "and" e "or" , ma la loro logica potrebbe facilmente diventare aggrovigliata e richiedere un debugging approfondito.

```
  1 false && true || echo false         # false
  2
  3 # Stesso risultato di
  4 ( false && true ) || echo false     # false
  5 # Ma *non*
  6 false && ( true || echo false )     # (non viene visualizzato niente)
  7
  8 #  Notate i raggruppamenti e la valutazione degli enunciati da sinistra a
destra
  9 #+ perché gli operatori logici "&&" e "||" hanno la stessa priorità.
 10
 11 #  È meglio evitare tali complessità, a meno che non sappiate cosa state
facendo
 12
 13 #  Grazie, S.C.
```

Vedi Esempio A-8 e Esempio 7-4 per un'illustrazione dell'uso di una **lista and / or** per la verifica di variabili

Capitolo 26. Array

Le versioni più recenti di Bash supportano gli array monodimensionali. Gli elementi dell'array posso essere inizializzati con la notazione **`variabile[xx]`**. In alternativa, uno script può introdurre un intero array con l'enunciato esplicito **`declare -a variabile`**. Per dereferenziare (cercare il contenuto di) un elemento dell'array, si usa la notazione *parentesi graffe*, vale a dire, **`${variabile[xx]}`**.

Esempio 26-1. Un semplice uso di array

```
 1 #!/bin/bash
 2
 3
 4 area[11]=23
 5 area[13]=37
 6 area[51]=UFO
 7
 8 # Non occorre che gli elementi dell'array siano consecutivi o contigui.
 9
10 # Alcuni elementi possono rimanere non inizializzati
11 # I "buchi" negli array sono permessi
12
13
14 echo -n "area[11] = "
15 echo ${area[11]}    #  sono necessarie le {parentesi graffe}
16
17 echo -n "area[13] = "
18 echo ${area[13]}
19
20 echo "Il contenuto di area[51] è ${area[51]}."
21
22 # Gli elementi non inizializzati vengono visualizzati come spazi.
23 echo -n "area[43] = "
24 echo ${area[43]}
25 echo "(area[43] non assegnato)"
26
27 echo
28
29 # Somma di due elementi dell'array assegnata ad un terzo
30 area[5]=`expr ${area[11]} + ${area[13]}`
31 echo "area[5] = area[11] + area[13]"
32 echo -n "area[5] = "
33 echo ${area[5]}
34
35 area[6]=`expr ${area[11]} + ${area[51]}`
36 echo "area[6] = area[11] + area[51]"
37 echo -n "area[6] = "
38 echo ${area[6]}
39 #  Questo assegnamento fallisce perché non è permesso sommare
40 #+ un intero con una stringa.
41
42 echo; echo; echo
43
44 # --------------------------------------------------------------------------
---
45 # Un altro array, "area2".
46 # Metodo di assegnamento alternativo...
47 # nome_array=( XXX YYY ZZZ ... )
48
49 area2=( zero uno due tre quattro )
50
51 echo -n "area2[0] = "
```

```
 52 echo ${area2[0]}
 53 #  Aha, indicizzazione in base zero (il primo elemento dell'array
 54 #+ è [0], non [1]).
 55
 56 echo -n "area2[1] = "
 57 echo ${area2[1]}    # [1] è il secondo elemento dell'array.
 58 # --------------------------------------------------------------------------
---
 59
 60 echo; echo; echo
 61
 62 # ------------------------------------------------
 63 # Ancora un altro array, "area3".
 64 # Ed un'altra modalità ancora di assegnamento...
 65 # nome_array=([xx]=XXX [yy]=YYY ...)
 66
 67 area3=([17]=diciassette [24]=ventiquattro)
 68
 69 echo -n "area3[17] = "
 70 echo ${area3[17]}
 71
 72 echo -n "area3[24] = "
 73 echo ${area3[24]}
 74 # ------------------------------------------------
 75
 76 exit 0
```

Bash consente le operazioni sugli arrray anche se questi non sono stati dichiarati tali esplicitamente.

```
  1 stringa=abcABC123ABCabc
  2 echo ${stringa[@]}              # abcABC123ABCabc
  3 echo ${stringa[*]}              # abcABC123ABCabc
  4 echo ${stringa[0]}              # abcABC123ABCabc
  5 echo ${stringa[1]}              # Nessun output!
  6                                 # Perchè?
  7 echo ${#stringa[@]}             # 1
  8                                 # Array di un solo elemento.
  9                                 # La stringa stessa.
 10
 11 # Grazie a Michael Zick per la precisazione.
```

Una volta ancora questo dimostra che le variabili Bash non sono tipizzate.

Esempio 26-2. Impaginare una poesia

```
  1 #!/bin/bash
  2 #  poem.sh: Visualizza in modo elegante una delle poesie preferite
  3 #+          dall'autore del documento.
  4
  5 # Righe della poesia (una strofa).
  6 Riga[1]="I do not know which to prefer,"
  7 Riga[2]="The beauty of inflections"
  8 Riga[3]="Or the beauty of innuendoes,"
  9 Riga[4]="The blackbird whistling"
 10 Riga[5]="Or just after."
 11
 12 # Attribuzione.
 13 Attrib[1]=" Wallace Stevens"
 14 Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
```

```
15 # La poesia è di Dominio Pubblico (copyright scaduto).
16
17 echo
18
19 for indice in 1 2 3 4 5    # Cinque righe.
20 do
21   printf "     %s\n" "${Riga[indice]}"
22 done
23
24 for indice in 1 2          # Attribuzione di due righe.
25 do
26   printf "          %s\n" "${Attrib[indice]}"
27 done
28
29 echo
30
31 exit 0
32
33 # Esercizio:
34 # ---------
35 #  Modificate lo script in modo che la poesia da visualizzare
36 #+ sia presa da un file dati.
```

Gli array hanno una sintassi loro propria ed anche i comandi e gli operatori standard di Bash posseggono opzioni specifiche adatte per l'uso degli array.

Esempio 26-3. Operazioni diverse sugli array

```
 1 #!/bin/bash
 2 # array-ops.sh: Un po' di divertimento con gli array.
 3
 4
 5 array=( zero uno due tre quattro cinque )
 6 #Elemento 0   1   2   3     4      5
 7
 8 echo ${array[0]}       #  zero
 9 echo ${array:0}        #  zero
10                        #  Espansione di parametro del primo elemento,
11                        #+ iniziando dalla posizione nr. 0 (1° carattere).
12 echo ${array:1}        #  ero
13                        #  Espansione di parametro del primo elemento,
14                        #+ iniziando dalla posizione nr. 1 (2° carattere).
15
16 echo "--------------"
17
18 echo ${#array[0]}      #  4
19                        #  Lunghezza del primo elemento dell'array.
20 echo ${#array}         #  4
21                        #  Lunghezza del primo elemento dell'array.
22                        #  (Notazione alternativa)
23
24 echo ${#array[1]}      #  3
25                        #  Lunghezza del secondo elemento dell'array.
26                        #  Gli array in Bash sono indicizzati in base zero.
27
28 echo ${#array[*]}      #  6
29                        #  Numero di elementi di array.
30 echo ${#array[@]}      #  6
31                        #  Numero di elementi di array.
32
```

```
33 echo "--------------"
34
35 array2=( [0]="primo elemento" [1]="secondo elemento" [3]="quarto elemento" )
36
37 echo ${array2[0]}      # primo elemento
38 echo ${array2[1]}      # secondo elemento
39 echo ${array2[2]}      #
40                        # Saltato durante l'inizializzazione, quindi nullo.
41 echo ${array2[3]}      # quarto elemento
42
43
44 exit 0
```

Con gli array funzionano anche molte delle normali operazioni stringa.

Esempio 26-4. Operazioni sulle stringhe negli array

```
 1 #!/bin/bash
 2 # array-strops.sh: Operazioni su stringhe negli array.
 3 # Script di Michael Zick.
 4 # Usato con il permesso dell'autore.
 5
 6 #  In generale, qualsiasi operazione stringa nella notazione ${nome ... }
 7 #+ può essere applicata a tutti gli elementi stringa presenti in un array
 8 #+ usando la notazione ${nome[@] ... } o ${nome[*] ...}.
 9
10
11 arrayZ=( uno due tre quattro cinque cinque )
12
13 echo
14
15 # Estrazione di sottostringa successiva
16 echo ${arrayZ[@]:0}     # uno due tre quattro cinque cinque
17                         # Tutti gli elementi.
18
19 echo ${arrayZ[@]:1}     # due tre quattro cinque cinque
20                         # Tutti gli elementi successivi ad elemento[0].
21
22 echo ${arrayZ[@]:1:2}   # due tre
23                         # Solo i due elementi successivi ad elemento[0].
24
25 echo "-----------------------"
26
27 #  Rimozione di sottostringa
28 #  Rimuove l'occorrenza più breve dalla parte iniziale della(e) stringa(he),
29 #+ dove sottostringa è un'espressione regolare.
30
31 echo ${arrayZ[@]#q*o}   # uno due tre cinque cinque
32                         # Controlla tutti gli elementi dell'array.
33                         # Verifica "quattro" e lo rimuove.
34
35 # L'occorrenza più lunga dalla parte iniziale della(e) stringa(he)
36 echo ${arrayZ[@]##t*e}  # uno due quattro cinque cinque
37                         # Controlla tutti gli elementi dell'array.
38                         # Verifica "tre" e lo rimuove.
39
40 # L'occorrenza più breve dalla parte finale della(e) stringa(he)
41 echo ${arrayZ[@]%r*e}   # uno due t quattro cinque cinque
42                         # Controlla tutti gli elementi dell'array.
43                         # Verifica "re" e lo rimuove.
44
```

```
# L'occorrenza più lunga dalla parte finale della(e) stringa(he)
echo ${arrayZ[@]%%t*e}  # uno due quattro cinque cinque
                        # Controlla tutti gli elementi dell'array.
                        # Verifica "tre" e lo rimuove.

echo "-----------------------"

# Sostituzione di sottostringa

# Rimpiazza la prima occorrenza di sottostringa con il sostituto
echo ${arrayZ[@]/cin/XYZ}   # uno due tre quattro XYZque XYZque
                            # Controlla tutti gli elementi dell'array.

# Sostituzione di tutte le occorrenze di sottostringa
echo ${arrayZ[@]//in/YY}    # uno due tre quattro cYYque cYYque
                            # Controlla tutti gli elementi dell'array.

# Cancellazione di tutte le occorrenze di sottostringa
# Non specificare la sostituzione significa 'cancellare'
echo ${arrayZ[@]//ci/}      # uno due tre quattro nque nque
                            # Controlla tutti gli elementi dell'array.

# Sostituzione delle occorrenze di sottostringa nella parte iniziale
echo ${arrayZ[@]/#ci/XY}    # uno due tre quattro XYnque XYnque
                            # Controlla tutti gli elementi dell'array.

# Sostituzione delle occorrenze di sottostringa nella parte finale
echo ${arrayZ[@]/%ue/ZZ}    # uno dZZ tre quattro cinqZZ cinqZZ
                            # Controlla tutti gli elementi dell'array.

echo ${arrayZ[@]/%o/XX}     # unXX due tre quattrXX cinque cinque
                            # Perché?

echo "-----------------------"


# Prima di passare ad awk (o altro) --
# Ricordate:
#   $( ... ) è la sostituzione di comando.
#   Le funzioni vengono eseguite come sotto-processi.
#   Le funzioni scrivono i propri output allo stdout.
#   L'assegnamento legge lo stdout della funzione.
#   La notazione nome[@] specifica un'operazione "for-each" (per-ogni).

nuovastr() {
    echo -n "!!!"
}

echo ${arrayZ[@]/%e/$(nuovastr)}
# uno du!!! tr!!! quattro cinqu!!! cinqu!!!
# Q.E.D:* L'azione di sostituzione è un 'assegnamento.'

#  Accesso "For-Each"
echo ${arrayZ[@]//*/$(nuovastr argomenti_opzionali)}
#  Ora, se Bash volesse passare semplicemente la stringa verificata come $0
#+ alla funzione da richiamare . . .

echo

exit 0

# * Quod Erat Demonstrandum: come volevasi dimostrare [N.d.T.]
```

Con la sostituzione di comando è possibile creare i singoli elementi di un array.

Esempio 26-5. Inserire il contenuto di uno script in un array

```
 1 #!/bin/bash
 2 # script-array.sh: Inserisce questo stesso script in un array.
 3 # Ispirato da una e-mail di Chris Martin (grazie!).
 4
 5 contenuto_script=( $(cat "$0") ) #  Registra il contenuto di questo script
($0)
 6                                   #+ in un array.
 7
 8 for elemento in $(seq 0 $((${#contenuto_script[@]} - 1)))
 9   do                #  ${#contenuto_script[@]}
10                     #+ fornisce il numero degli elementi di un array.
11                     #
12                     #  Domanda:
13                     #  Perché è necessario  seq 0?
14                     #  Provate a cambiarlo con seq 1.
15   echo -n "${contenuto_script[$elemento]}"
16                     # Elenca tutti i campi dello script su una sola riga.
17   echo -n " -- "    # Usa " -- " come separatore di campo.
18 done
19
20 echo
21
22 exit 0
23
24 # Esercizio:
25 # ---------
26 #  Modificate lo script in modo che venga visualizzato
27 #+ nella sua forma originale,
28 #+ completa di spazi, interruzioni di riga, ecc.
```

Nel contesto degli array, alcuni builtin di Bash assumono un significato leggermente diverso. Per esempio, unset cancella gli elementi dell'array o anche un intero array.

Esempio 26-6. Alcune proprietà particolari degli array

```
 1 #!/bin/bash
 2
 3 declare -a colori
 4 #  Tutti i comandi successivi presenti nello script tratteranno
 5 #+ la variabile "colori" come un array.
 6
 7 echo "Inserisci i tuoi colori preferiti (ognuno separato da uno spazio)."
 8
 9 read -a colori    #  Inserite almeno 3 colori per mettere alla prova le
10                   #+ funzionalità che seguono.
11 #  Opzione speciale del comando 'read',
12 #+ che consente l'assegnamento degli elementi di un array.
13
14 echo
15
16 conta_elementi=${#colori[@]}
17 # Sintassi speciale per ricavare il numero di elementi di un array.
18 #     conta_elementi=${#colori[*]} anche in questo modo.
19 #
20 #  La variabile "@" permette la suddivisione delle parole, anche se
```

```
all'interno
 21 #+ degli apici (estrae le variabili separate da spazi).
 22
 23 #  Corrisponde al comportamento di "$@" e "$*"
 24 #+ nei parametri posizionali.
 25
 26 indice=0
 27
 28 while [ "$indice" -lt "$conta_elementi" ]
 29 do    # Elenca tutti gli elementi dell' array.;
 30   echo ${colori[$indice]}
 31   let "indice = $indice + 1"
 32 done
 33 # Ogni elemento dell'array viene visualizzato su una riga singola.
 34 # Se non vi piace, utilizzate  echo -n "${colori[$indice]} "
 35 #
 36 # La stessa cosa utilizzando un ciclo "for":
 37 #   for i in "${colori[@]}"
 38 #   do
 39 #     echo "$i"
 40 #   done
 41 # (Grazie, S.C.)
 42
 43 echo
 44
 45 #  Ancora, elenco di tutti gli elementi dell'array utilizzando, però, un
 46 #+ metodo più elegante.
 47   echo ${colori[@]}         # anche echo ${colori[*]}.
 48
 49 echo
 50
 51 # Il comando "unset" cancella gli elementi di un array, o l'intero array.
 52 unset colori[1]             # Cancella il secondo elemento dell' array.
 53                             # Stesso effetto di colori[1]=
 54 echo  ${colori[@]}          # Elenca ancora l'array. Manca il secondo
elemento.
 55
 56 unset colori                # Cancella l'intero array.
 57                             #  Anche: unset colori[*] e
 58                             #+ unset colori[@].
 59 echo; echo -n "Colori cancellati."
 60 echo ${colori[@]}           # Visualizza ancora l'array, ora vuoto.
 61
 62 exit 0
```

Come si è visto nell'esempio precedente, sia **${nome_array[@]}** che **${nome_array[*]}** fanno riferimento a tutti gli elementi dell'array. Allo stesso modo, per ottenere il numero degli elementi di un array si usa sia **${#nome_array[@]}** che **${#nome_array[*]}**. **${#nome_array}** fornisce la lunghezza (numero di caratteri) di **${nome_array[0]}**, il primo elemento dell'array.

Esempio 26-7. Array vuoti ed elementi vuoti

```
  1 #!/bin/bash
  2 # empty-array.sh
  3
  4 #  Grazie a Stephane Chazelas, per l'esempio originario,
  5 #+ e a Michael Zick, per averlo ampliato.
  6
  7
  8 # Un array vuoto non è la stessa cosa di un array composto da elementi
```

```
vuoti.
  9
 10 array0=( primo secondo terzo )
 11 array1=( '' )   # "array1" contiene un elemento vuoto.
 12 array2=( )      # Nessun elemento . . . "array2" è vuoto.
 13
 14 echo
 15 ElencaArray ()
 16 {
 17 echo
 18 echo "Elementi in array0:  ${array0[@]}"
 19 echo "Elementi in array1:  ${array1[@]}"
 20 echo "Elementi in array2:  ${array2[@]}"
 21 echo
 22 echo "Lunghezza del primo elemento di array0 = ${#array0}"
 23 echo "Lunghezza del primo elemento di array1 = ${#array1}"
 24 echo "Lunghezza del primo elemento di array2 = ${#array2}"
 25 echo
 26 echo "Numero di elementi di array0 = ${#array0[*]}"  # 3
 27 echo "Numero di elementi di array1 = ${#array1[*]}"  # 1  (Sorpresa!)
 28 echo "Numero di elementi di array2 = ${#array2[*]}"  # 0
 29 }
 30
 31 # ==================================================================
 32
 33 ElencaArray
 34
 35 # Proviamo ad incrementare gli array
 36
 37 # Aggiunta di un elemento ad un array.
 38 array0=( "${array0[@]}" "nuovo1" )
 39 array1=( "${array1[@]}" "nuovo1" )
 40 array2=( "${array2[@]}" "nuovo1" )
 41
 42 ElencaArray
 43
 44 # oppure
 45 array0[${#array0[*]}]="nuovo2"
 46 array1[${#array1[*]}]="nuovo2"
 47 array2[${#array2[*]}]="nuovo2"
 48
 49 ElencaArray
 50
 51 # Quando sono modificati in questo modo, gli array sono come degli 'stack'
 52 # L'operazione precedente rappresenta un 'push'
 53 # L''altezza' dello stack è:
 54 altezza=${#array2[@]}
 55 echo
 56 echo "Altezza dello stack array2 = $altezza"
 57
 58 # Il 'pop' è:
 59 unset array2[${#array2[@]}-1]     #  Gli array hanno indici in base zero
 60 altezza=${#array2[@]}           #+ vale a dire che il primo elemento ha
indice 0
 61 echo
 62 echo "POP"
 63 echo "Nuova altezza dello stack array2 = $altezza"
 64
 65 ElencaArray
 66
 67 # Elenca solo gli elemnti 2do e 3zo dell'array0
 68 da=1            # Numerazione in base zero
```

```
 69 a=2             #
 70 array3=( ${array0[@]:1:2} )
 71 echo
 72 echo "Elementi dell'array3:  ${array3[@]}"
 73
 74 # Funziona come una stringa (array di caratteri)
 75 # Provate qualche altro tipo di "stringa"
 76
 77 # Sostituzione:
 78 array4=( ${array0[@]/secondo/2do} )
 79 echo
 80 echo "Elementi dell'array4:  ${array4[@]}"
 81
 82 # Sostituzione di ogni occorrenza della stringa con il carattere jolly
 83 array5=( ${array0[@]//nuovo?/vecchio} )
 84 echo
 85 echo "Elementi dell'array5:  ${array5[@]}"
 86
 87 # Proprio quando stavate per prenderci la mano . . .
 88 array6=( ${array0[@]#*nuovo} )
 89 echo # Questo potrebbe sorprendervi.
 90 echo "Elementi dell'array6:  ${array6[@]}"
 91
 92 array7=( ${array0[@]#nuovo1} )
 93 echo # Dopo l'array6 questo non dovrebbe più stupirvi.
 94 echo "Elementi dell'array7:  ${array7[@]}"
 95
 96 # Che assomiglia moltissimo a . . .
 97 array8=( ${array0[@]/nuovo1/} )
 98 echo
 99 echo "Elementi dell'array8:  ${array8[@]}"
100
101 #  Quindi, cosa possiamo dire a questo proposito?
102
103 #  Le operazioni stringa vengono eseguite su ognuno
104 #+ degli elementi presenti in var[@] in sequenza.
105 #  Quindi : Bash supporta le operazioni su vettore stringa.
106 #+ Se il risultato è una stringa di lunghezza zero,
107 #+ quell'elemento scompare dall'assegnamento risultante.
108
109 #  Domanda: queste stringhe vanno usate con il quoting forte o debole?
110
111 zap='nuovo*'
112 array9=( ${array0[@]/$zap/} )
113 echo
114 echo "Elementi dell'array9:  ${array9[@]}"
115
116 # Proprio quando pensavate di essere a cavallo . . .
117 array10=( ${array0[@]#$zap} )
118 echo
119 echo "Elementi dell'array10:  ${array10[@]}"
120
121 # Confrontate array7 con array10.
122 # Confrontate array8 con array9.
123
124 # Risposta: con il quoting debole.
125
126 exit 0
```

La relazione tra **${nome_array[@]}** e **${nome_array[*]}** è analoga a quella tra $@ e $*. Questa potente notazione degli array ha molteplici impieghi.

```
 1 # Copiare un array.
 2 array2=( "${array1[@]}" )
 3 # oppure
 4 array2="${array1[@]}"
 5
 6 # Aggiunta di un elemento ad un array.
 7 array=( "${array[@]}" "nuovo elemento" )
 8 # oppure
 9 array[${#array[*]}]="nuovo elemento"
10
11 # Grazie, S.C.
```

L'operazione di inizializzazione **array=(elemento1 elemento2 ... elementoN**), con l'aiuto della sostituzione di comando, permette di inserire in un array il contenuto di un file di testo.

```
 1 #!/bin/bash
 2
 3 nomefile=file_esempio
 4
 5 #            cat file_esempio
 6 #
 7 #            1 a b c
 8 #            2 d e fg
 9
10
11 declare -a array1
12
13 array1=( `cat "$nomefile" | tr '\n' ' '`)  #  Carica il contenuto
14 #         Visualizza il file allo stdout   #+ di $nomefile in array1.
15 #
16 #  array1=( `cat "$nomefine" | tr '\n' ' '`)
17 #                             cambia i ritorni a capo presenti nel file
in spazi.
18 #  Non necessario perchè Bash effettua la suddivisione delle parole
19 #+ che modifica i ritorni a capo in spazi.
20
21 echo ${array1[@]}            # Visualizza il contenuto dell'array.
22 #                              1 a b c 2 d e fg
23 #
24 #  Ogni "parola" separata da spazi presente nel file
25 #+ è stata assegnata ad un elemento dell'array.
26
27 conta_elementi=${#array1[*]}
28 echo $conta_elementi         # 8
```

Uno scripting intelligente consente di aggiungere ulteriori operazioni sugli array.

Esempio 26-8. Initializzare gli array

```
 1 #! /bin/bash
 2 # array-assign.bash
 3
 4 #  Le operazioni degli array sono specifiche di Bash,
 5 #+ quindi il nome dello script deve avere il suffisso ".bash".
 6
 7 # Copyright (c) Michael S. Zick, 2003, Tutti i diritti riservati.
 8 # Licenza: Uso illimitato in qualsiasi forma e per qualsiasi scopo.
 9 # Versione: $ID$
10
11 # Chiarimenti e commenti aggiuntivi di William Park.
```

```
12
13 #  Basato su un esempio fornito da Stephane Chazelas,
14 #+ apparso nel libro: Guida avanzata di bash scripting.
15
16 # Formato dell'output del comando 'times':
17 # CPU Utente <spazio> CPU Sistema
18 # CPU utente di tutti i processi <spazio> CPU sistema di tutti i processi
19
20 #  Bash possiede due modi per assegnare tutti gli elementi di un array
21 #+ ad un nuovo array.
22 #  Nelle versioni Bash 2.04, 2.05a e 2.05b.
23 #+ entrambi i metodi inseriscono gli elementi 'nulli'
24 #  Alle versioni più recenti può aggiungersi un ulteriore assegnamento
25 #+ purché, per tutti gli array, sia mantenuta la relazione [indice]=valore.
26
27 #  Crea un array di grandi dimensioni utilizzando un comando interno,
28 #+ ma andrà bene qualsiasi cosa che permetta di creare un array
29 #+ di diverse migliaia di elementi.
30
31 declare -a grandePrimo=( /dev/* )
32 echo
33 echo 'Condizioni: Senza quoting, IFS preimpostato, Tutti gli elementi'
34 echo "Il numero di elementi dell'array è ${#grandePrimo[@]}"
35
36 # set -vx
37
38
39
40 echo
41 echo '- - verifica: =( ${array[@]} ) - -'
42 times
43 declare -a grandeSecondo=( ${grandePrimo[@]} )
44 #                        ^                  ^
45 times
46
47 echo
48 echo '- - verifica: =${array[@]} - -'
49 times
50 declare -a grandeTerzo=${grandePrimo[@]}
51 # Questa volta niente parentesi.
52 times
53
54 #  Il confronto dei risultati dimostra che la seconda forma, evidenziata
55 #+ da Stephane Chazelas, è da tre a quattro volte più veloce.
56 #
57 #  Spiega William Park:
58 #+ L'array grandeSecondo viene inizializzato come stringa singola,
59 #+ mentre grandeTerzo viene inizializzato elemento per elemento.
60 #  Quindi, in sostanza, abbiamo:
61 #                  grandeSecondo=( [0]="... ... ..." )
62 #                  grandeTerzo=( [0]="..." [1]="..." [2]="..." ... )
63
64
65 #  Nei miei esempi esplicativi, continuerò ad utilizzare la prima forma
66 #+ perché penso serva ad illustrare meglio quello che avviene.
67
68 #  In realtà, porzioni di codice di miei esempi possono contenere
69 #+ la seconda forma quando è necessario velocizzare l'esecuzione.
70
71 # MSZ: Scusate le precedenti sviste.
72
73 #  Nota:
```

```
74 #  ----
75 #  Gli enunciati "declare -a" alle righe 31 e 43
76 #+ non sarebbero strettamente necessari perchè sono impliciti
77 #+ nell'assegnamento nella forma  Array=( ... ).
78 #  Tuttavia, l'eliminazione di queste dichiarazioni rallenta
79 #+ l'esecuzione delle successive sezioni dello script.
80 #  Provate e vedete cosa succede.
81
82 exit 0
```


L'aggiunta del superfluo enunciato **declare -a** nella dichiarazione di un array può velocizzare l'esecuzione delle successive operazioni sullo stesso array.

Esempio 26-9. Copiare e concatenare array

```
 1 #! /bin/bash
 2 # CopyArray.sh
 3 #
 4 # Script di Michael Zick.
 5 # Usato con il permesso dell'autore.
 6
 7 #  Come "Passare per Nome & restituire per Nome"
 8 #+ ovvero "Costruirsi il proprio enunciato di assegnamento".
 9
10
11 CpArray_Mac() {
12
13 # Costruttore dell'enunciato di assegnamento
14
15     echo -n 'eval '
16     echo -n "$2"                    # Nome di destinazione
17     echo -n '=( ${'
18     echo -n "$1"                    # Nome di origine
19     echo -n '[@]} )'
20
21 # Si sarebbe potuto fare con un solo comando.
22 # E' solo una questione di stile.
23 }
24
25 declare -f CopiaArray               # Funzione "Puntatore"
26 CopiaArray=CpArray_Mac              # Costruttore dell'ennuciato
27
28 Enfatizza()
29 {
30
31 # Enfatizza l'array di nome $1.
32 # (Lo sposa all'array contenente "veramente fantastico".)
33 # Risultato nell'array di nome $2.
34
35     local -a TMP
36     local -a esagerato=( veramente fantastico )
37
38     $($CopiaArray $1 TMP)
39     TMP=( ${TMP[@]} ${esagerato[@]} )
40     $($CopiaArray TMP $2)
41 }
42
43 declare -a prima=( Lo scripting di Bash avanzato )
44 declare -a dopo
45
46 echo "Array iniziale = ${prima[@]}"
```

```
47
48 Enfatizza prima dopo
49
50 echo "Array finale = ${dopo[@]}"
51
52 # Troppo esagerato?
53
54 echo "Cos'è ${dopo[@]:4:2}?"
55
56 declare -a modesto=( ${dopo[@]:0:2} "è" ${dopo[@]:4:2} )
57 #                     - estrazione di sottostringhe -
58
59 echo "Array modesto = ${modesto[@]}"
60
61 # Cos'è successo a 'prima' ?
62
63 echo "Array iniziale = ${prima[@]}"
64
65 exit 0
```

Esempio 26-10. Ancora sulla concatenazione di array

```
 1 #! /bin/bash
 2 # array-append.bash
 3
 4 # Copyright (c) Michael S. Zick, 2003, Tutti i diritti riservati.
 5 # Licenza: Uso illimitato in qualsiasi forma e per qualsiasi scopo.
 6 # Versione: $ID$
 7 #
 8 # Impaginazione leggermente modificata da M.C.
 9
10
11 # Le operazioni degli array sono specifiche di Bash.
12 # La /bin/sh originaria UNIX non ne possiede di equivalenti.
13
14
15 #  Collagate con una pipe l'output dello script a 'more'
16 #+ in modo che non scorra completamente sullo schermo.
17
18
19 # Inizializzazione abbreviata.
20 declare -a array1=( zero1 uno1 due1 )
21 # Inizializzazione dettagliata ([1] non viene definito).
22 declare -a array2=( [0]=zero2 [2]=due2 [3]=tre2 )
23
24 echo
25 echo "- Conferma che l'array è stato inizializzato per singolo elemento. -"
26 echo "Numero di elementi: 4"        # Codificato a scopo illustrativo.
27 for (( i = 0 ; i < 4 ; i++ ))
28 do
29     echo "Elemento [$i]: ${array2[$i]}"
30 done
31 # Vedi anche il codice d'esempio più generale in basics-reviewed.bash.
32
33
34 declare -a dest
35
36 # Combina (accodando) i due array in un terzo.
37 echo
38 echo 'Condizioni: Senza quoting, IFS preimpostato, operatore Array-intero'
39 echo '- Elementi non definiti assenti, indici non mantenuti. -'
```

```
 40 # Gli elementi non definiti non esistono; non vengono inseriti.
 41
 42 dest=( ${array1[@]} ${array2[@]} )
 43 # dest=${array1[@]}${array2[@]}     # Risultati strani, probabilmente un
bug.
 44
 45 # Ora visualizziamo il risultato.
 46 echo
 47 echo "- - Verifica dell'accodamento dell'array - -"
 48 cnt=${#dest[@]}
 49
 50 echo "Numero di elementi: $cnt"
 51 for (( i = 0 ; i < cnt ; i++ ))
 52 do
 53     echo "Elemento [$i]: ${dest[$i]}"
 54 done
 55
 56 # (Doppio) Assegnamento di un intero array ad un elemento di un altro array.
 57 dest[0]=${array1[@]}
 58 dest[1]=${array2[@]}
 59
 60 # Visualizzazione del risultato.
 61 echo
 62 echo "- - Verifica dell'array modificato - -"
 63 cnt=${#dest[@]}
 64
 65 echo "Numero di elementi: $cnt"
 66 for (( i = 0 ; i < cnt ; i++ ))
 67 do
 68     echo "Elemento [$i]: ${dest[$i]}"
 69 done
 70
 71 # Esame del secondo elemento modificato.
 72 echo
 73 echo '- - Riassegnazione e visualizzazione del secondo elemento - -'
 74
 75 declare -a subArray=${dest[1]}
 76 cnt=${#subArray[@]}
 77
 78 echo "Numero di elementi: $cnt"
 79 for (( i = 0 ; i < cnt ; i++ ))
 80 do
 81     echo "Elemento [$i]: ${subArray[$i]}"
 82 done
 83
 84 #  L'assegnamento di un intero array ad un singolo elemento
 85 #+ di un altro array, utilizzando la notazione '=${ ... }',
 86 #+ ha trasformato l'array da assegnare in una stringa,
 87 #+ con gli elementi separati da uno spazio (il primo carattere di IFS).
 88
 89 # Se gli elementi d'origine non avessero contenuto degli spazi . . .
 90 # Se l'array d'origine non fosse stato inizializzato in modo dettagliato . .
.
 91 # Allora come risultato si sarebbe ottenuto la struttura dell'array
d'origine.
 92
 93 # Ripristino con il secondo elemento modificato.
 94 echo
 95 echo "- - Visualizzazione dell'elemento ripristinato - -"
 96
 97 declare -a subArray=( ${dest[1]} )
 98 cnt=${#subArray[@]}
```

```
 99
100 echo "Numero di elementi: $cnt"
101 for (( i = 0 ; i < cnt ; i++ ))
102 do
103     echo "Elemento [$i]: ${subArray[$i]}"
104 done
105 echo '- - Non fate affidamento su questo comportamento. - -'
106 echo '- - Potrebbe divergere nelle versioni di Bash - -'
107 echo '- - precedenti alla 2.05b - -'
108
109 # MSZ: Mi scuso per qualsiasi confusa spiegazione fatta in precedenza.
110
111 exit 0
```

--

Gli array consentono la riscrittura, in forma di script di shell, di vecchi e familiari algoritmi. Se questa sia necessariamente una buona idea, è lasciato al lettore giudicare.

Esempio 26-11. Un vecchio amico: *Il Bubble Sort*

```
  1 #!/bin/bash
  2 # bubble.sh: Ordinamento a bolle.
  3
  4 # Ricordo l'algoritmo dell'ordinamento a bolle. In questa particolare
versione..
  5
  6 #  Ad ogni passaggio successivo lungo l'array che deve essere ordinato,
  7 #+ vengono confrontati due elementi adiacenti e scambiati se non ordinati.
  8 #  Al termine del primo passaggio, l'elemento "più pesante" è sprofondato
  9 #+ nell'ultima posizione dell'array. Al termine del secondo passaggio, il
 10 #+ rimanente elemento "più pesante" si trova al penultimo posto. E così via.
 11 #+ Questo significa che ogni successivo passaggio deve attraversare una
 12 #+ porzione minore di array. Noterete, quindi, un aumento della velocità
 13 #+ di visualizzazione dopo ogni passaggio.
 14
 15
 16 scambio()
 17 {
 18   # Scambia due membri dell'array.
 19   local temp=${Paesi[$1]} #  Variabile per la memorizzazione temporanea
 20                           #+ dell'elemento che deve essere scambiato.
 21   Paesi[$1]=${Paesi[$2]}
 22   Paesi[$2]=$temp
 23
 24   return
 25 }
 26
 27 declare -a Paesi  #  Dichiara l'array,
 28                   #+ in questo caso facoltativo perché viene inizializzato
 29                   #+ successivamente.
 30
 31 #  È consentito suddividere l'inizializzazione di un array su più righe
 32 #+ utilizzando il carattere di escape (\)?
 33 #  Sì.
 34
 35 Paesi=(Olanda Ucraina Zaire Turchia Russia Yemen Siria \
 36 Brasile Argentina Nicaragua Giappone Messico Venezuela Grecia Inghilterra \
 37 Israele Peru Canada Oman Danimarca Galles Francia Kenya \
 38 Xanadu Qatar Liechtenstein Ungheria)
```

```
 39
 40 # "Xanadu" è il luogo mitico dove, secondo Coleridge,
 41 #+"Kubla Khan fece un duomo di delizia fabbricare".
 42
 43
 44 clear                  # Pulisce lo schermo prima di iniziare
l'elaborazione.
 45
 46 echo "0: ${Paesi[*]}"  # Elenca l'intero array al passaggio 0.
 47
 48 numero_di_elementi=${#Paesi[@]}
 49 let "confronti = $numero_di_elementi - 1"
 50
 51 conto=1 # Numero di passaggi.
 52
 53 while [ "$confronti" -gt 0 ]               # Inizio del ciclo esterno
 54 do
 55
 56   indice=0  # L'indice viene azzerato all'inizio di ogni passaggio.
 57
 58   while [ "$indice" -lt "$confronti" ]      # Inizio del ciclo interno
 59   do
 60     if [ ${Paesi[$indice]} \> ${Paesi[`expr $indice + 1`]} ]
 61     #  Se non ordinato...
 62     #  Ricordo che \> è l'operatore di confronto ASCII
 63     #+ usato all'interno delle parantesi quadre singole.
 64
 65     #  if [[ ${Paesi[$indice]} > ${Paesi[`expr $indice + 1`]} ]]
 66     #+ anche in questa forma.
 67     then
 68         scambio $indice `expr $indice + 1`  # Scambio.
 69     fi
 70     let "indice += 1"
 71   done # Fine del ciclo interno
 72
 73
 74 # ------------------------------------------------------------------------
 75 #  Paulo Marcel Coelho Aragao suggerisce una più semplice alternativa
 76 #+ utilizzando i cicli for.
 77 #
 78 # for (( ultimo = $numero_di_elementi - 1 ; ultimo > 1 ; ultimo-- ))
 79 # do
 80 #     for (( i = 0 ; i < ultimo ; i++ ))
 81 #     do
 82 #         [[ "${Paesi[$i]}" > "${Paesi[$((i+1))]}" ]] \
 83 #             && scambio $i $((i+1))
 84 #     done
 85 # done
 86 # ------------------------------------------------------------------------
 87
 88
 89 let "confronti -= 1" #  Poiché l'elemento "più pesante" si è depositato in
 90                      #+ fondo, è necessario un confronto in meno ad ogni
 91                      #+ passaggio.
 92
 93 echo
 94 echo "$conto: ${Paesi[@]}"  #  Visualizza la situazione dell'array al
termine
 95                            #+ di ogni passaggio.
 96 echo
 97 let "conto += 1"            # Incrementa il conteggio dei passaggi.
 98
```

```
 99 done                          # Fine del ciclo esterno
100                               # Completato.
101
102 exit 0
```

--

È possibile annidare degli array in altri array?

```
 1 #!/bin/bash
 2 #  Array "annidato".
 3
 4 #  Esempio fornito da Michael Zick,
 5 #+ con correzioni e chiarimenti di William Park.
 6
 7 UnArray=( $(ls --inode --ignore-backups --almost-all \
 8         --directory --full-time --color=none --time=status \
 9         --sort=time -l ${PWD} ) )  # Comandi e opzioni.
10
11 # Gli spazi sono significativi . . . quindi non si deve usare il quoting.
12
13 SubArray=( ${UnArray[@]:11:1}  ${UnArray[@]:6:5} )
14 #  Questo array è formato da sei elementi:
15 #+     SubArray=( [0]=${UnArray[11]} [1]=${UnArray[6]} [2]=${UnArray[7]}
16 #      [3]=${UnArray[8]} [4]=${UnArray[9]} [5]=${UnArray[10]} )
17 #
18 #  In Bash gli array sono liste collegate (circolarmente)
19 #+ del tipo stringa (char *).
20 #  Quindi, non si tratta veramente di un array annidato,
21 #+ è il suo comportamento che è simile.
22
23 echo "Directory corrente e data dell'ultima modifica:"
24 echo "${SubArray[@]}"
25
26 exit 0
```

--

Gli array annidati in combinazione con la referenziazione indiretta creano affascinanti possibilità

Esempio 26-12. Array annidati e referenziazioni indirette

```
 1 #!/bin/bash
 2 # embedded-arrays.sh
 3 # Array annidati e referenziazione indiretta.
 4
 5 # Script di Dennis Leeuw.
 6 # Usato con il permesso dell'autore.
 7 # Modificato dall'autore di questo documento.
 8
 9
10 ARRAY1=(
11         VAR1_1=valore11
12         VAR1_2=valore12
13         VAR1_3=valore13
14 )
15
16 ARRAY2=(
17         VARIABILE="test"
```

```
 18         STRINGA="VAR1=valore1 VAR2=valore2 VAR3=valore3"
 19         ARRAY21=${ARRAY1[*]}
 20 )       # L'ARRAY1 viene inserito in questo secondo array.
 21
 22 function visualizza () {
 23         PREC_IFS="$IFS"
 24         IFS=$'\n'       #  Per visualizzare ogni elemento dell'array
 25                         #+ su una riga diversa.
 26         TEST1="ARRAY2[*]"
 27         local ${!TEST1} # Provate a vedere cosa succede cancellando questa
riga.
 28         #  Referenziazione indiretta.
 29         #  Questo rende i componenti di $TEST1
 30         #+ accessibili alla funzione.
 31
 32
 33         #  A questo punto, vediamo cosa abbiamo fatto.
 34         echo
 35         echo "\$TEST1 = $TEST1"       #  Solo il nome della variabile.
 36         echo; echo
 37         echo "{\$TEST1} = ${!TEST1}"  #  Contenuto della variabile.
 38                                       #  Questo è ciò che fa la
 39                                       #+ referenziazione indiretta.
 40         echo
 41         echo "-----------------------------------------"; echo
 42         echo
 43
 44
 45         # Visualizza la variabile
 46         echo "Variabile VARIABILE: $VARIABILE"
 47
 48         # Visualizza un elemento stringa
 49         IFS="$PREC_IFS"
 50         TEST2="STRINGA[*]"
 51         local ${!TEST2}      # Referenziazione indiretta (come prima).
 52         echo "Elemento stringa VAR2: $VAR2 da STRINGA"
 53
 54         # Visualizza un elemento dell'array
 55         TEST2="ARRAY21[*]"
 56         local ${!TEST2}      # Referenziazione indiretta (come prima).
 57         echo "Elemento VAR1_1 dell'array: $VAR1_1 da ARRAY21"
 58 }
 59
 60 visualizza
 61 echo
 62
 63 exit 0
 64
 65 #   Come fa notare l'autore,
 66 #+ "lo script può facilmente essere espanso per ottenere gli hash
 67 #+  anche nella shell bash."
 68 #   Esercizio per i lettori (difficile): implementate questa funzionalità.
```

--

Gli array permettono l'implementazione, in versione di script di shell, del *Crivello di Eratostene*. Naturalmente, un'applicazione come questa, che fa un uso così intensivo di risorse, in realtà dovrebbe essere scritta in un linguaggio compilato, come il C. Sotto forma di script, la sua esecuzione è atrocemente lenta.

Esempio 26-13. Applicazione complessa di array: *Crivello di Eratostene*

```
  1 #!/bin/bash
  2 # sieve.sh
  3
  4 # Crivello di Eratostene
  5 # Antico algoritmo per la ricerca di numeri primi.
  6
  7 # L'esecuzione è di due ordini di grandezza
  8 # più lenta dell'equivalente programma scritto in C.
  9
 10 LIMITE_INFERIORE=1       # Si inizia da 1.
 11 LIMITE_SUPERIORE=1000    # Fino a 1000.
 12 # (Potete impostarlo ad un valore più alto...  se avete tempo a
disposizione.)
 13
 14 PRIMO=1
 15 NON_PRIMO=0
 16
 17 let META=LIMITE_SUPERIORE/2
 18 # Ottimizzazione:
 19 # È necessario verificare solamente la metà dei numeri.
 20
 21
 22 declare -a Primi
 23 # Primi[] è un array.
 24
 25
 26 inizializza ()
 27 {
 28 # Inizializza l'array.
 29
 30 i=$LIMITE_INFERIORE
 31 until [ "$i" -gt "$LIMITE_SUPERIORE" ]
 32 do
 33   Primi[i]=$PRIMO
 34   let "i += 1"
 35 done
 36 # Assumiamo che tutti gli elementi dell'array siano colpevoli (primi)
 37 # finché non verrà provata la loro innocenza (non primi).
 38 }
 39
 40 visualizza_primi ()
 41 {
 42 # Visualizza gli elementi dell'array Primi[] contrassegnati come primi.
 43
 44 i=$LIMITE_INFERIORE
 45
 46 until [ "$i" -gt "$LIMITE_SUPERIORE" ]
 47 do
 48
 49   if [ "${Primi[i]}" -eq "$PRIMO" ]
 50   then
 51     printf "%8d" $i
 52     # 8 spazi per numero producono delle colonne belle ed uniformi.
 53   fi
 54
 55   let "i += 1"
 56
 57 done
 58
 59 }
```

```
 60
 61 vaglia () # Identifica i numeri non primi.
 62 {
 63
 64 let i=$LIMITE_INFERIORE+1
 65 # Sappiamo che 1 è primo, quindi iniziamo da 2.
 66
 67 until [ "$i" -gt "$LIMITE_SUPERIORE" ]
 68 do
 69
 70 if [ "${Primi[i]}" -eq "$PRIMO" ]
 71 #  Non si preoccupa di vagliare i numeri già verificati (contrassegnati come
 72 #+ non-primi).
 73 then
 74
 75   t=$i
 76
 77   while [ "$t" -le "$LIMITE_SUPERIORE" ]
 78   do
 79     let "t += $i "
 80     Primi[t]=$NON_PRIMO
 81     # Segna come non-primi tutti i multipli.
 82   done
 83
 84 fi
 85
 86   let "i += 1"
 87 done
 88
 89
 90 }
 91
 92
 93 # Invoca le funzioni sequenzialmente.
 94 inizializza
 95 vaglia
 96 visualizza_primi
 97 # Questa è quella che si chiama programmazione strutturata.
 98
 99 echo
100
101 exit 0
102
103
104
105 # ----------------------------------------------- #
106 # Il codice oltre la riga precedente non viene eseguito.
107
108 # Questa versione migliorata del Crivello, di Stephane Chazelas,
109 # esegue il compito un po' più velocemente.
110
111 #  Si deve invocare con un argomento da riga di comando (il limite dei
112 #+ numeri primi).
113
114 LIMITE_SUPERIORE=$1            # Da riga di comando.
115 let META=LIMITE_SUPERIORE/2    # Metà del numero massimo.
116
117 Primi=( '' $(seq $LIMITE_SUPERIORE) )
118
119 i=1
120 until (( ( i += 1 ) > META ))  #  È sufficiente verificare solo la metà dei
121                                #+ numeri.
```

```
122 do
123   if [[ -n $Primi[i] ]]
124   then
125     t=$i
126     until (( ( t += i ) > LIMITE_SUPERIORE ))
127     do
128       Primi[t]=
129     done
130   fi
131 done
132 echo ${Primi[*]}
133
134 exit 0
```

Si confronti questo generatore di numeri primi, basato sugli array, con uno alternativo che non li utilizza, Esempio A-17.

--

Gli array si prestano, entro certi limiti, a simulare le strutture di dati per le quali Bash non ha un supporto nativo.

Esempio 26-14. Simulare uno stack push-down

```
 1 #!/bin/bash
 2 # stack.sh: simulazione di uno stack push-down
 3
 4 #  Simile ad uno stack di CPU, lo stack  push-down registra i dati
 5 #+ sequenzialmente, ma li rilascia in ordine inverso, last-in first-out
 6 #+ (l'ultimo inserito è il primo prelevato).
 7
 8 BP=100            #  Base Pointer (puntatore alla base) dello stack (array).
 9                   #  Inizio dall'elemento 100.
10
11 SP=$BP            #  Stack Pointer (puntatore allo stack).
12                   #  Viene inizializzato alla "base" (fondo) dello stack.
13
14 Dato=             #  Contenuto di una locazione dello stack.
15                   #  Deve essere una variabile locale,
16                   #+ a causa della limitazione del valore di ritorno di
17                   #+ una funzione.
18
19 declare -a stack
20
21
22 push()            # Pone un dato sullo stack.
23 {
24 if [ -z "$1" ]    # Niente da immettere?
25 then
26   return
27 fi
28
29 let "SP -= 1"     # Riposiziona lo stack pointer.
30 stack[$SP]=$1
31
32 return
33 }
34
35 pop()                    # Preleva un dato dallo stack.
```

```
{
Dato=                    # Svuota la variabile.

if [ "$SP" -eq "$BP" ]   # Lo stack è vuoto?
then
  return
fi                       #  Questo evita anche che SP oltrepassi il 100,
                         #+ cioè, impedisce la "fuga" dallo stack.

Dato=${stack[$SP]}
let "SP += 1"            # Riposiziona lo stack pointer.
return
}

situazione()             # Permette di verificare quello che sta avvenendo.
{
echo "-----------------------------------"
echo "RAPPORTO"
echo "Stack Pointer = $SP"
echo "Appena dopo che \""$Dato"\" è stato prelevato dallo stack."
echo "-----------------------------------"
echo
}


# ======================================================
# Ora un po' di divertimento.

echo

# Vedete se riuscite a prelevare qualcosa da uno stack vuoto.
pop
situazione

echo

push rifiuto
pop
situazione    # Rifiuto inserito, rifiuto tolto.

valore1=23; push $valore1
valore2=skidoo; push $valore2
valore3=FINALE; push $valore3

pop           # FINALE
situazione
pop           # skidoo
situazione
pop           # 23
situazione    # Last-in, first-out!

#  Fate attenzione che lo stack pointer si decrementa ad ogni push,
#+ e si incrementa ad ogni pop.

echo
# ======================================================


# Esercizi:
# --------

# 1)  Modificate la funzione "push()" in modo che consenta l'immissione
```

```
 98 #   + nello stack di più dati con un'unica chiamata.
 99
100 # 2)  Modificate la funzione "pop()" in modo che consenta di prelevare
101 #   + dallo stack più dati con un'unica chiamata.
102
103 # 3)  Utilizzando questo script come base di partenza, scrivete un programma
104 #   + per una calcolatrice a 4 funzioni basate sullo stack.
105
106 exit 0
107
108 #  N.d.T. - Si è preferito lasciare inalterati i termini, in quanto
109 #+ appartenenti al linguaggio di programmazione Assembly. La traduzione è
110 #+ stata posta tra parentesi o nei commenti.
```

--

Elaborate manipolazioni dei "subscript" [1] degli array possono richiedere l'impiego di variabili intermedie. In progetti dove questo è richiesto, si consideri, una volta ancora, l'uso di un linguaggio di programmazione più potente, come Perl o C.

Esempio 26-15. Applicazione complessa di array: *Esplorare strane serie matematiche*

```
  1 #!/bin/bash
  2
  3 # I celebri "numeri Q" di Douglas Hofstadter
  4
  5 # Q(1) = Q(2) = 1
  6 # Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), per n>2
  7
  8 #  È una successione di interi "caotica" con comportamento strano e
  9 #+ non prevedibile.
 10 # I primi 20 numeri della serie sono:
 11 # 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12
 12
 13 #  Vedi il libro di Hofstadter, "Goedel, Escher, Bach: un'Eterna Ghirlanda
 14 #+ Brillante", p. 149, ff. (Ed. italiana Adelphi - terza edizione -
settembre
 15 #+ 1985 [N.d.T.])
 16
 17
 18 LIMITE=100      # Numero di termini da calcolare
 19 AMPIEZZARIGA=20 # Numero di termini visualizzati per ogni riga
 20
 21 Q[1]=1          # I primi due numeri della serie sono 1.
 22 Q[2]=1
 23
 24 echo
 25 echo "Numeri Q [$LIMITE termini]:"
 26 echo -n "${Q[1]} "             # Visualizza i primi due termini.
 27 echo -n "${Q[2]} "
 28
 29 for ((n=3; n <= $LIMITE; n++))  # ciclo con condizione in stile C.
 30 do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  per n>2
 31 #  È necessario suddividere l'espressione in termini intermedi,
 32 #  perché Bash non è in grado di gestire molto bene la matematica complessa
 33 #+ degli array.
 34
 35   let "n1 = $n - 1"        # n-1
 36   let "n2 = $n - 2"        # n-2
 37
```

```
 38   t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
 39   t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]
 40
 41   T0=${Q[t0]}              # Q[n - Q[n-1]]
 42   T1=${Q[t1]}              # Q[n - Q[n-2]]
 43
 44 Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
 45 echo -n "${Q[n]} "
 46
 47 if [ `expr $n % $AMPIEZZARIGA` -eq 0 ]    # Ordina l'output.
 48 then   #    modulo
 49   echo # Suddivide le righe in blocchi ordinati.
 50 fi
 51
 52 done
 53
 54 echo
 55
 56 exit 0
 57
 58 #  Questa è un'implementazione iterativa dei numeri Q.
 59 #  L'implementazione più intuitiva, che utilizza la ricorsività, è lasciata
 60 #+ come esercizio.
 61 #  Attenzione: calcolare la serie ricorsivamente richiede un tempo *molto*
lungo.
```

--

Bash supporta solo gli array monodimensionali, tuttavia un piccolo stratagemma consente di simulare quelli multidimensionali.

Esempio 26-16. Simulazione di un array bidimensionale, con suo successivo rovesciamento

```
  1 #!/bin/bash
  2 # twodim.sh: Simulazione di un array bidimensionale.
  3
  4 # Un array monodimensionale è formato da un'unica riga.
  5 # Un array bidimensionale registra le righe sequenzialmente.
  6
  7 Righe=5
  8 Colonne=5
  9 # Array 5 X 5.
 10
 11 declare -a alfa      # alfa [Righe] [Colonne];
 12                      # Dichiarazione non necessaria. Perché?
 13
 14 inizializza_alfa ()
 15 {
 16 local rc=0
 17 local indice
 18
 19 for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
 20 do     # Se preferite, potete utilizzare simboli differenti.
 21   local riga=`expr $rc / $Colonne`
 22   local colonna=`expr $rc % $Righe`
 23   let "indice = $riga * $Righe + $colonna"
 24   alfa[$indice]=$i
 25 # alfa[$riga][$colonna]
 26   let "rc += 1"
 27 done
```

```
 28
 29 #  Sarebbe stato più semplice
 30 #   declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
 31 #+ ma, per così dire, si sarebbe perso il "gusto" dell'array bi-
dimensionale.
 32 }
 33
 34 visualizza_alfa ()
 35 {
 36 local riga=0
 37 local indice
 38
 39 echo
 40
 41 while [ "$riga" -lt "$Righe" ] #  Visualizza in ordine di precedenza di
riga:
 42 do                             #+ variano le colonne
 43                                #+ mentre la riga (ciclo esterno) non cambia.
 44   local colonna=0
 45
 46   echo -n "       "            #  Allinea l'array "quadrato" con quello
ruotato.
 47
 48   while [ "$colonna" -lt "$Colonne" ]
 49   do
 50    let "indice = $riga * $Righe + $colonna"
 51    echo -n "${alfa[indice]} "  # alfa[$riga][$colonna]
 52    let "colonna += 1"
 53   done
 54
 55   let "riga += 1"
 56   echo
 57
 58 done
 59
 60 # L'analogo più semplice è
 61 #   echo ${alfa[*]} | xargs -n $Colonne
 62
 63 echo
 64 }
 65
 66 filtra ()     # Elimina gli indici negativi dell'array.
 67 {
 68
 69 echo -n "  "  # Fornisce l'inclinazione.
 70               # Spiegate perché.
 71
 72 if [[ "$1" -ge 0 &&  "$1" -lt "$Righe" && "$2" -ge 0 && "$2" -lt "$Colonne"
]]
 73 then
 74  let "indice = $1 * $Righe + $2"
 75  # Ora lo visualizza ruotato.
 76  echo -n " ${alfa[indice]}"
 77  #           alfa[$riga][$colonna]
 78 fi
 79
 80 }
 81
 82
 83
 84
 85 ruota ()  #  Ruota l'array di 45 gradi --
```

```
 86 {         #+ facendo "perno" sul suo angolo inferiore sinistro.
 87 local riga
 88 local colonna
 89
 90 for (( riga = Righe; riga > -Righe; riga-- ))
 91 do          # Passa l'array in senso inverso. Perchè?
 92
 93   for (( colonna = 0; colonna < Colonne; colonna++ ))
 94   do
 95
 96     if [ "$riga" -ge 0 ]
 97     then
 98         let "t1 = $colonna - $riga"
 99         let "t2 = $colonna"
100     else
101        let "t1 = $colonna"
102        let "t2 = $colonna + $riga"
103     fi
104
105     filtra $t1 $t2   # Elimina gli indici negativi dell'array. Perchè?
106   done
107
108   echo; echo
109
110 done
111
112 #  La rotazione è ispirata agli esempi (pp. 143-146) presenti in
113 #+ "Advanced C Programming on the IBM PC," di Herbert Mayer
114 #+ (vedi bibliografia).
115 #  Questo solo per dimostrare che molto di quello che si può fare con il C
116 #+ può essere fatto con lo scripting di shell.
117
118 }
119
120
121 #--------------- E ora, che lo spettacolo inizi.-------------#
122 inizializza_alfa  # Inizializza l'array.
123 visualizza_alfa   # Lo visualizza.
124 ruota             # Lo ruota di 45 gradi in senso antiorario.
125 #------------------------------------------------------------#
126
127 exit 0
128
129 # Si tratta di una simulazione piuttosto macchinosa, per non dire
inelegante.
130 #
131 # Esercizi:
132 # ---------
133 # 1)  Riscrivete le funzioni di inizializzazione e visualizzazione
134 #     in maniera più intuitiva ed elegante.
135 #
136 # 2)  Illustrate come operano le funzioni di rotazione dell'array.
137 #     Suggerimento: pensate alle implicazioni di una indicizzazione
138 #     inversa dell'array.
139 #
140 # 3)  Riscrivete lo script in modo da gestire un array non quadrato,
141 #     come uno di dimensioni 6 X 4.
142 #     Cercate di minimizzare la "distorsione" quando l'array viene ruotato.
```

Un array bidimensionale equivale essenzialmente ad uno monodimensionale, ma con modalità aggiuntive per poter individuare, ed eventualmente manipolare, il singolo elemento in base alla sua posizione per "riga" e "colonna".

Per una dimostrazione ancor più elaborata di simulazione di un array bidimensionale, vedi Esempio A-11.

Note

[1] Con questo termine, nel linguaggio C, vengono chiamati gli indici degli array (N.d.T.)

Capitolo 27. File

file di avvio (startup)

Questi file contengono gli alias e le variabili d'ambiente che vengono rese disponibili a Bash, in esecuzione come shell utente, e a tutti gli script Bash invocati dopo l'inizializzazione del sistema.

`/etc/profile`

valori predefiniti del sistema, la maggior parte dei quali inerenti all'impostazione dell'ambiente (tutte le shell di tipo Bourne, non solo Bash [1])

`/etc/bashrc`

funzioni e alias di sistema per Bash

`$HOME/.bash_profile`

impostazioni d'ambiente predefinite di Bash specifiche per il singolo utente. Si trova in ogni directory home degli utenti (è il corrispettivo locale di `/etc/profile`)

`$HOME/.bashrc`

file init Bash specifico per il singolo utente. Si trova in ogni directory home degli utenti (è il corrispettivo locale di `/etc/bashrc`). Solo le shell interattive e gli script utente leggono questo file. In Appendice J viene riportato un esempio di un file `.bashrc`.

file di arresto (logout)

`$HOME/.bash_logout`

file di istruzioni specifico dell'utente. Si trova in ogni directory home degli utenti. Dopo l'uscita da una shell di login (Bash), vengono eseguiti i comandi presenti in questo file.

Note

[1] Questo non è valido per **csh**, **tcsh** e per tutte le altre shell non imparentate o non derivanti dalla

classica shell Bourne (**sh**).

Capitolo 28. /dev e /proc

Sommario

Una tipica macchina Linux, o UNIX, possiede due directory con scopi specifici: `/dev` e `/proc`.

28.1. /dev

La directory `/dev` contiene l'elenco di tutti i *dispositivi* fisici che possono o meno essere presenti nel hardware. [1] Le partizioni di un hard disk contenenti il/i filesystem montato/i si trovano in `/dev`, come un semplice df può mostrare.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6               495876    222748    247527  48% /
 /dev/hda1                50755      3887     44248   9% /boot
 /dev/hda8               367013     13262    334803   4% /home
 /dev/hda5              1714416   1123624    503704  70% /usr
```

Tra l'altro, la directory `/dev` contiene anche i dispositivi di *loopback*, come `/dev/loop0`. Un dispositivo di loopback rappresenta un espediente che permette l'accesso ad un file ordinario come se si trattasse di un dispositivo a blocchi. [2] In questo modo si ha la possibilità di montare un intero filesystem all'interno di un unico, grande file. Vedi Esempio 13-7 e Esempio 13-6.

In `/dev` sono presenti anche alcuni altri file con impieghi specifici, come `/dev/null`, `/dev/zero`, `/dev/urandom`, `/dev/sda1`, `/dev/udp`, e `/dev/tcp`.

Ad esempio:

Per montare una memoria flash USB, si aggiunga la riga seguente nel file `/etc/fstab`. [3]

```
  1 /dev/sda1    /mnt/memoriaflash    auto    noauto,user,noatime    0 0
```

(Vedi anche Esempio A-22.)

Quando viene eseguito un comando sul file di pseudo-dispositivo `/dev/tcp/$host/$porta`, Bash apre una connessione TCP al *socket* associato. [4]

Ottenere l'ora da `nist.gov`:

```
bash$ cat </dev/tcp/time.nist.gov/13
53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

```

```

[L'esempio precedente è stato fornito da Mark.]

Scaricare un URL:

```
bash$ exec 5<>/dev/tcp/www.slashdot.org/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
bash$ cat <&5
```

[Grazie a Mark e Mihai Maties.]

Esempio 28-1. Uso di `/dev/tcp` per la verifica di una connessione

```
  1 #!/bin/bash
  2 # dev-tcp.sh: /dev/tcp redirezione per il controllo della connessione
Internet.
  3
  4 # Script di Troy Engel.
  5 # Utilizzato con il permesso dell'autore.
  6
  7 HOST_TCP=www.slashdot.org
  8 PORTA_TCP=80   # La porta 80 è usata da http.
  9
 10 # Tentativo di connessione. (Abbastanza simile a un 'ping.')
 11 echo "HEAD / HTTP/1.0" >/dev/tcp/${HOST_TCP}/${PORTA_TCP}
 12 MIOEXIT=$?
 13
 14 : << SPIEGAZIONE
 15 Se bash è stata compilata con l'opzione --enable-net-redirections, ha la
 16 capacità di utilizzare uno speciale dispositivo a caratteri per redirigere
 17 sia TCP che UDP. Queste redirezioni vengono usate alla stessa identica
maniera
 18 degli STDIN/STDOUT/STDERR. I valori per il dispositivo /dev/tcp sono 30,36:
 19
 20   mknod /dev/tcp c 30 36
 21
 22 >Dalla bash reference:
 23 /dev/tcp/host/port
 24     Se host è un nome valido o un indirizzo Internet, e port un numero
 25 intero di una porta o il nome di un servizio, Bash tenta di aprire una
 26 connessione TCP al socket corrispondente.
 27 SPIEGAZIONE
 28
 29
 30 if [ "X$MIOEXIT" = "X0" ]; then
 31   echo "Connessione riuscita. Codice d'uscita: $MIOEXIT"
 32 else
 33   echo "Connessione fallita. Codice d'uscita: $MIOEXIT"
 34 fi
 35
 36 exit $MIOEXIT
```

Note

[1] I file presenti in `/dev` forniscono i punti di mount per i dispositivi fisici o virtuali. Queste

registrazioni occupano pochissimo spazio su disco.

Alcuni dispositivi, come `/dev/null`, `/dev/zero` e `/dev/urandom` sono virtuali. Non corrispondono, quindi, ad alcun dispositivo fisico ed esistono solo a livello software.

[2] Un *dispositivo a blocchi* legge e/o scrive i dati in spezzoni, o blocchi, a differenza di un *dispositivo a caratteri* che accede ai dati un carattere alla volta. Esempi di dispositivi a blocchi sono l'hard disk e il CD ROM. Un esempio di dispositivo a caratteri è la tastiera.

[3] Naturalmente, il punto di mount `/mnt/memoriaflash` dev'essere già stato creato. In caso contrario, come utente root, **mkdir /mnt/memoriaflash**.

Il "montaggio" effettivo della memoria viene effettuato tramite il comando: **mount /mnt/memoriaflash**

[4] Un *socket* è un nodo di comunicazione associato ad una specifica porta I/O. Consente il traferimento di dati tra i dispositivi hardware della stessa macchina, tra macchine della stessa rete, tra macchine appartenenti a reti diverse e, naturalmente, tra macchine dislocate in posti differnti dell'Internet.

28.2. `/proc`

La directory `/proc`, in realtà, è uno pseudo-filesystem. I file in essa contenuti rispecchiano il sistema correntemente in esecuzione, i *processi* del kernel, ed informazioni e statistiche su di essi.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md



bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
```

```
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
ERR:          0


bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse
running use aveq

   3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0
111550 644030
   3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
   3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
   3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
   ...



bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119



bash$ cat /proc/apm
1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?
```

Gli script di shell possono ricavare dati da alcuni dei file presenti in `/proc`. [1]

```
 1 FS=iso                      #  Il supporto per il filesystem ISO è
 2                             #+ abilitato nel kernel?
 3 grep $FS /proc/filesystems  # iso9660
```

```
 1 versione_kernel=$( awk '{ print $3 }' /proc/version )
```

```
 1 CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )
 2
 3 if [ $CPU = Pentium ]
 4 then
 5   esegui_dei_comandi
 6   ...
 7 else
 8   esegui_altri_comandi
 9   ...
10 fi
```

```
 1 filedisp="/proc/bus/usb/devices"
 2 USB1="Spd=12"
 3 USB2="Spd=480"
 4
 5
 6 veloc_bus=$(grep Spd $filedisp | awk '{print $9}')
 7
 8 if [ "$veloc_bus" = "$USB1" ]
 9 then
10   echo "Trovata porta USB 1.1."
11   # Comandi inerenti alla porta USB 1.1.
12 fi
```

La directory `/proc` contiene delle sottodirectory con strani nomi numerici. Ognuno di questi nomi traccia l'ID di processo di tutti i processi correntemente in esecuzione. All'interno di ognuna di queste sottodirectory, vi è un certo numero di file contenenti utili informazioni sui processi corrispondenti. I file `stat` e `status` contengono statistiche continuamente aggiornate del processo, il file `cmdline` gli argomenti da riga di comando con i quali il processo è stato invocato e il file `exe` è un link simbolico al percorso completo del processo chiamante. Di tali file ve ne sono anche altri (pochi), ma quelli elencati sembrano essere i più interessanti dal punto di vista dello scripting.

Esempio 28-2. Trovare il processo associato al PID

```
  1 #!/bin/bash
  2 #  pid-identifier.sh: Fornisce il percorso completo del processo associato
al
  3 #+ pid.
  4
  5 ARGNUM=1  # Numero di argomenti attesi dallo script.
  6 E_ERR_ARG=65
  7 E_ERR_PID=66
  8 E_ERR_PROCESSO=67
  9 E_ERR_PERMESSO=68
 10 FILEPROC=exe
 11
 12 if [ $# -ne $ARGNUM ]
 13 then
 14   echo "Utilizzo: `basename $0` numero PID" >&2  # Messaggio d'errore
>stderr.
 15   exit $E_ERR_ARG
 16 fi
 17
 18 pidnum=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
 19 #  Controlla il pid nell'elenco di "ps", campo nr.1.
 20 #  Quindi si accerta che sia il processo effettivo, non quello invocato
dallo
 21 #+ script stesso.
 22 # L'ultimo "grep $1" scarta questa possibilità.
 23 if [ -z "$pidnum" ]  #  Se, anche dopo il filtraggio, il risultato è una
 24                      #+ stringa di lunghezza zero,
 25 then                 #  significa che nessun processo in esecuzione
 26                      #+ corrisponde al pid dato.
 27   echo "Il processo non è in esecuzione."
 28   exit $E_ERR_PROCESSO
 29 fi
 30
 31 # In alternativa:
 32 #   if ! ps $1 > /dev/null 2>&1
 33 #   then             # nessun processo in esecuzione corrisponde al pid
dato.
 34 #     echo "Il processo non è in esecuzione."
 35 #     exit $E_ERR_PROCESSO
 36 #    fi
 37
 38 # Per semplificare l'intera procedura, si usa "pidof".
 39
 40
 41 if [ ! -r "/proc/$1/$FILEPROC" ]  # Controlla i permessi in lettura.
 42 then
 43   echo "Il processo $1 è in esecuzione, ma..."
 44   echo "Non ho il permesso di lettura su /proc/$1/$FILEPROC."
 45   exit $E_ERR_PERMESSO  #  Un utente ordinario non può accedere ad alcuni
 46                         #+ file di /proc.
```

```
fi

# Le due ultime verifiche possono essere sostituite da:
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' non è un segnale, ma
                                      # verifica la possibilità
                                      # di inviare un segnale al processo.
#    then echo "Il PID non esiste o non sei il suo proprietario" >&2
#    exit $E_ERR_PID
#    fi


file_exe=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
#  Oppure   file_exe=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
#  /proc/numero-pid/exe è un link simbolico
#+ al nome completo del processo chiamante.

if [ -e "$file_exe" ]  # Se /proc/numero-pid/exe esiste...
then                   # esiste anche il corrispondente processo.
  echo "Il processo nr.$1 è stato invocato da $file_exe."
else
  echo "Il processo non è in esecuzione."
fi


# Questo elaborato script si potrebbe *quasi* sostituire con
# ps ax | grep $1 | awk '{ print $5 }'
# Questa forma, però, non funzionerebbe...
# perché il quinto campo di 'ps' è  l'argv[0] del processo,
# non il percorso del file eseguibile.
#
# Comunque, entrambi i seguenti avrebbero funzionato.
#       find /proc/$1/exe -printf '%l\n'
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'

# Commenti aggiuntivi di Stephane Chazelas.

exit 0
```

Esempio 28-3. Stato di una connessione

```
#!/bin/bash

NOMEPROC=pppd        # Demone ppp
NOMEFILEPROC=status  # Dove guardare.
NONCONNESSO=65
INTERVALLO=2         # Aggiorna ogni 2 secondi.

pidnum=$( ps ax | grep -v "ps ax" | grep -v grep | grep $NOMEPROC \
| awk '{ print $1 }' )

# Ricerca del numero del processo di 'pppd', il 'demone ppp'.
# Occorre eliminare le righe del processo generato dalla ricerca stessa.
#
#  Comunque, come ha evidenziato Oleg Philon,
#+ lo si sarebbe potuto semplificare considerevolmente usando "pidof".
#  pidnum=$( pidof $NOMEPROC )
#
#  Morale della favola:
#  Quando una sequenza di comandi diventa troppo complessa, cercate una
#+ scorciatoia.
```

```
21
22
23 if [ -z "$pidnum" ]   #  Se non c'è il pid, allora il processo non è
24                       #+ in esecuzione.
25 then
26   echo "Non connesso."
27   exit $NONCONNESSO
28 else
29   echo "Connesso."; echo
30 fi
31
32 while [ true ]       # Ciclo infinito. Qui lo script può essere migliorato.
33 do
34
35   if [ ! -e "/proc/$pidnum/$NOMEFILEPROC" ]
36   # Finché il processo è in esecuzione, esiste il file "status".
37   then
38       echo "Disconnesso."
39       exit $NONCONNESSO
40   fi
41
42 netstat -s | grep "packets received"  # Per avere alcune statistiche.
43 netstat -s | grep "packets delivered"
44
45
46   sleep $INTERVALLO
47   echo; echo
48
49 done
50
51 exit 0
52
53 # Così com'è, lo script deve essere terminato con Control-C.
54
55 #    Esercizi:
56 #    ---------
57 #    Migliorate lo script in modo che termini alla pressione del tasto "q".
58 #    Rendete lo script più amichevole inserendo altre funzionalità
59
```


In generale, è pericoloso *scrivere* nei file presenti in `/proc` perché questo potrebbe portare alla corruzione del filesystem o al crash della macchina.

Note

[1] Alcuni comandi di sistema, come procinfo, free, vmstat, lsdev, e uptime svolgono lo stesso compito.

Capitolo 29. Zero e Null

`/dev/zero` e `/dev/null`

Usi di `/dev/null`

> Si pensi a `/dev/null` come a un "buco nero". Equivale, quasi, ad un file in sola scrittura. Tutto quello che vi viene scritto scompare per sempre. I tentativi di leggerene o di

visualizzarne il contenuto non danno alcun risultato. Ciò nonostante, `/dev/null` può essere piuttosto utile sia da riga di comando che negli script.

Sopprimere lo `stdout`.

```
  1 cat $nomefile >/dev/null
  2 # Il contenuto del file non verrà elencato allo stdout.
```

Sopprimere lo `stderr` (da Esempio 12-3).

```
  1 rm $nomestrano 2>/dev/null
  2 #           Così i messaggi d'errore [stderr] vengono "sotterrati".
```

Sopprimere gli output da *entrambi*, `stdout` e `stderr`.

```
  1 cat $nomefile 2>/dev/null >/dev/null
  2 #  Se "$nomefile" non esiste, come output non ci sarà alcun
messaggio d'errore.
  3 #  Se "$nomefile" esiste, il suo contenuto non verrà elencato allo
stdout.
  4 #  Quindi, la riga di codice precedente, in ogni caso, non dà alcun
risultato.
  5 #
  6 #  Ciò può rivelarsi utile in situazioni in cui è necessario
verificare il
  7 #+ codice di ritorno di un comando, ma non si desidera visualizzarne
l'output.
  8 #
  9 # cat $nomefile &>/dev/null
 10 #     anche in questa forma, come ha sottolineato Baris Cicek.
```

Cancellare il contenuto di un file, preservando il file stesso ed i rispettivi permessi (da Esempio 2-1 e Esempio 2-3):

```
  1 cat /dev/null > /var/log/messages
  2 #  : > /var/log/messages   ha lo stesso effetto e non genera un
nuovo processo.
  3
  4 cat /dev/null > /var/log/wtmp
```

Svuotare automaticamente un file di log (ottimo specialmente per trattare quei disgustosi "cookie" inviati dai siti commerciali del Web):

Esempio 29-1. Evitare i cookie

```
  1 if [ -f ~/.netscape/cookies ]  # Se esiste, lo cancella.
  2 then
  3   rm -f ~/.netscape/cookies
  4 fi
  5
  6 ln -s /dev/null ~/.netscape/cookies
  7 #  Tutti i cookie vengono ora spediti nel buco nero, invece di
essere salvati
  8 #+ su disco.
```

Usi di `/dev/zero`

Come `/dev/null`, anche `/dev/zero` è uno pseudo file, ma in realtà contiene null (zeri binari, non del genere ASCII). Un output scritto nel file scompare, ed è abbastanza difficile leggere i null reali contenuti in `/dev/zero`, sebbene questo possa essere fatto con od o con un editor esadecimale. L'uso principale di `/dev/zero` è quello di creare un file fittizio inizializzato, della dimensione specificata, da usare come file di scambio (swap) temporaneo.

Esempio 29-2. Impostare un file di swap usando `/dev/zero`

```
 1 #!/bin/bash
 2 # Creare un file di swap.
 3
 4 UID_ROOT=0         # Root ha $UID 0.
 5 E_ERR_UTENTE=65    # Non root?
 6
 7 FILE=/swap
 8 DIMENSIONEBLOCCO=1024
 9 BLOCCHIMIN=40
10 SUCCESSO=0
11
12
13 # Questo script deve essere eseguito da root.
14 if [ "$UID" -ne "$UID_ROOT" ]
15 then
16   echo; echo "Devi essere root per eseguire questo script."; echo
17   exit $E_ERR_UTENTE
18 fi
19
20
21 blocchi=${1:-$BLOCCHIMIN}    #  Imposta a 40 blocchi il valore
predefinito, se
22                              #+ non viene specificato diversamente
da riga di
23                              #+ comando.
24 # Equivale al seguente blocco di codice.
25 # ----------------------------------------------------
26 # if [ -n "$1" ]
27 # then
28 #   blocchi=$1
29 # else
30 #   blocchi=$BLOCCHIMIN
31 # fi
32 # ----------------------------------------------------
33
34
35 if [ "$blocchi" -lt $BLOCCHIMIN ]
36 then
37   blocchi=$BLOCCHIMIN       # La dimensione deve essere di almeno 40
blocchi.
38 fi
39
40
41 echo "Creazione di un file di swap della dimensione di $bloccchi
blocchi (KB)."
42 dd if=/dev/zero of=$FILE bs=$DIMENSIONEBLOCCO count=$blocchi  #
Pone il file a
43                                                              #+
zero.
44
45 mkswap $FILE $blocchi             # Lo designa come file di swap.
```

```
 46 swapon $FILE                       # Attiva il file di swap.
 47
 48 echo "Il file di swap è stato creato ed attivato."
 49
 50 exit $SUCCESSO
```

Un'altra applicazione di `/dev/zero` è quella di "svuotare" un file della dimensione specificata da usare per uno scopo specifico, come montare un filesystem su un dispositivo di loopback (vedi Esempio 13-7) o per la cancellazione di sicurezza di un file (vedi Esempio 12-50).

Esempio 29-3. Creare un ramdisk

```
  1 #!/bin/bash
  2 # ramdisk.sh
  3
  4 #  Un "ramdisk" è un segmento della memoria RAM
  5 #+ che si comporta come se fosse un filesystem.
  6 #  Presenta il vantaggio di un accesso velocissimo (tempo di
lettura/scrittura)
  7 #  Svantaggi: volatilità, perdita di dati al riavvio o in caso di
mancanza di
  8 #+            corrente elettrica, meno RAM disponibile al sistema.
  9 #
 10 #  Cos'ha di buono un ramdisk?
 11 #  Tenere una serie di dati di grandi dimensioni, come una tabella o
un
 12 #+ dizionario, su un ramdisk ne velocizza la consultazione, perché
l'accesso
 13 #+ alla memoria è molto più veloce di un accesso al disco.
 14
 15
 16 E_NON_ROOT=70                  # Deve essere eseguito da root.
 17 NOME_ROOT=root
 18
 19 MOUNTPT=/mnt/ramdisk
 20 DIMENSIONE=2000                # 2K blocchi (modificare in base alle
esigenze)
 21 DIMENSIONEBLOCCO=1024          # 1K (1024 byte)
 22 DISPOSITIVO=/dev/ram0          # Primo dispositivo ram
 23
 24 nomeutente=`id -nu`
 25 if [ "$nomeutente" != "$NOME_ROOT" ]
 26 then
 27   echo "Devi essere root per eseguire \"`basename $0`\"."
 28   exit $E_NON_ROOT
 29 fi
 30
 31 if [ ! -d "$MOUNTPT" ]         #  Verifica se già esiste il punto di
mount,
 32 then                          #+ in modo che non ci sia un errore
se lo script
 33   mkdir $MOUNTPT              #+ viene eseguito più volte.
 34 fi
 35
 36 dd if=/dev/zero of=$DISPOSITIVO count=$DIMENSIONE
bs=$DIMENSIONEBLOCCO
 37                               #  Pone il dispositivo RAM a zero.
 38                               #  Perchè questa operazione è
necessaria?
```

```
 39 mke2fs $DISPOSITIVO            #  Crea, su di esso, un filesystem di
tipo ext2.
 40 mount $DISPOSITIVO $MOUNTPT    #  Lo monta.
 41 chmod 777 $MOUNTPT             #  Abilita l'accesso al ramdisk da
parte di un
 42                                #+ utente ordinario.
 43                                #  Tuttavia, si deve essere root per
smontarlo.
 44
 45 echo "\"$MOUNTPT\" ora è disponibile all'uso."
 46 #  Il ramdisk è accessibile, per la registrazione di file, anche ad
un utente
 47 #+ ordinario.
 48
 49 #  Attenzione, il ramdisk è volatile e il contenuto viene perso
 50 #+ in caso di riavvio del PC o mancanza di corrente.
 51 #  Copiate tutto quello che volete salvare in una directory
regolare.
 52
 53 # Dopo un riavvio, rieseguite questo script per reimpostare il
ramdisk.
 54 # Rifare il mount di /mnt/ramdisk senza gli altri passaggi è
inutile.
 55
 56 #  Opportunamente modificato, lo script può essere invocato in
 57 #+ /etc/rc.d/rc.local per impostare automaticamente un ramdisk in
fase di boot.
 58 #  Potrebbe essere appropriato, ad esempio, su un server database.
 59
 60 exit 0
```

Capitolo 30. Debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Brian Kernighan

La shell Bash non possiede alcun debugger e neanche comandi o costrutti specifici per il debugging. [1] Gli errori di sintassi o le errate digitazioni generano messaggi d'errore criptici che, spesso, non sono di alcun aiuto per correggere uno script che non funziona.

Esempio 30-1. Uno script errato

```
 1 #!/bin/bash
 2 # ex74.sh
 3
 4 # Questo è uno script errato.
 5 # Ma dove sarà mai l'errore?
 6
 7 a=37
 8
 9 if [$a -gt 27 ]
10 then
11   echo $a
```

```
12 fi
13
14 exit 0
```

Output dello script:

```
./ex74.sh: [37: command not found
```

Cosa c'è di sbagliato nello script precedente (suggerimento: dopo **if**)?

Esempio 30-2. Parola chiave mancante

```
 1 #!/bin/bash
 2 # missing-keyword.sh: Che messaggio d'errore verrà generato?
 3
 4 for a in 1 2 3
 5 do
 6   echo "$a"
 7 # done     #  La necessaria parola chiave 'done', alla riga 7,
 8            #+ è stata commentata.
 9
10 exit 0
```

Output dello script:

```
missing-keyword.sh: line 10: syntax error: unexpected end of file
```

È da notare che il messaggio d'errore *non* necessariamente si riferisce alla riga in cui questo si verifica, ma a quella dove l'interprete Bash si rende finalmente conto della sua presenza.

I messaggi d'errore, nel riportare il numero di riga di un errore di sintassi, potrebbero ignorare le righe di commento presenti nello script.

E se uno script funziona, ma non dà i risultati attesi? Si tratta del fin troppo familiare errore logico.

Esempio 30-3. test24, un altro script errato

```
 1 #!/bin/bash
 2
 3 #  Si suppone che questo script possa cancellare tutti i file della
 4 #+ directory corrente i cui nomi contengono degli spazi.
 5 #  Non funziona.
 6 #  Perché?
 7
 8
 9 bruttonome=`ls | grep ' '`
10
11 # Provate questo:
12 # echo "$bruttonome"
13
14 rm "$bruttonome"
15
16 exit 0
```

Si cerchi di scoprire cos'è andato storto in Esempio 30-3 decommentando la riga **echo "$bruttonome"**. Gli enunciati echo sono utili per vedere se quello che ci si aspetta è veramente quello che si è ottenuto.

In questo caso particolare, **rm "$bruttonome"** non dà il risultato desiderato perché non si sarebbe dovuto usare `$bruttonome` con il quoting. Averlo collocato tra apici significa assegnare a **rm** un unico argomento (verifica un solo nome di file). Una parziale correzione consiste nel togliere gli apici a `$bruttonome` ed impostare `$IFS` in modo che contenga solo il ritorno a capo, **IFS=$'\n'**. Esistono, comunque, modi più semplici per ottenere il risultato voluto.

```
1 # Metodi corretti per cancellare i file i cui nomi contengono spazi.
2 rm *\ *
3 rm *" "*
4 rm *' '*
5 # Grazie. S.C.
```

Riepilogo dei sintomi di uno script errato,

1. Comparsa del messaggio "syntax error", oppure
2. Va in esecuzione, ma non funziona come dovrebbe (errore logico).
3. Viene eseguito, funziona come ci si attendeva, ma provoca pericolosi effetti collaterali (bomba logica).

Gli strumenti per la correzione di script non funzionanti comprendono

1. gli enunciati echo posti in punti cruciali dello script, per tracciare le variabili ed avere così un quadro di quello che sta avvenendo.
2. l'uso del filtro tee nei punti critici per verificare i processi e i flussi di dati.
3. lanciare lo script con le opzioni `-n -v -x`

   sh -n nomescript verifica gli errori di sintassi senza dover eseguire realmente lo script. Equivale ad inserire nello script **set -n** o **set -o noexec**. È da notare che alcuni tipi di errori di sintassi possono eludere questa verifica.

   sh -v nomescript visualizza ogni comando prima della sua esecuzione. Equivale ad inserire nello script **set -v** o **set -o verbose**.

   Le opzioni `-n` e `-v` funzionano bene insieme. **sh -nv nomescript** fornisce una verifica sintattica dettagliata.

   sh -x nomescript visualizza il risultato di ogni comando, ma in modo abbreviato. Equivale ad inserire nello script **set -x** o **set -o xtrace**.

   Inserire **set -u** o **set -o nounset** nello script permette la sua esecuzione visualizzando, però, il messaggio d'errore "unbound variable" ogni volta che si cerca di usare una variabile non dichiarata.

4. L'uso di una funzione "assert", per verificare una variabile o una condizione, in punti critici dello script. (È un'idea presa a prestito dal C).

   Esempio 30-4. Verificare una condizione con "assert"

```
  1 #!/bin/bash
  2 # assert.sh
  3
  4 assert ()                 #  Se la condizione è falsa,
  5 {                        #+ esce dallo script con un messaggio
d'errore.
  6   E_ERR_PARAM=98
  7   E_ASSERT_FALLITA=99
  8
  9
 10   if [ -z "$2" ]          # Non sono stati passati abbastanza
parametri.
 11   then
 12     return $E_ERR_PARAM   # Non fa niente.
 13   fi
 14
 15   numriga=$2
 16
 17   if [ ! $1 ]
 18   then
 19     echo "Assert \"$1\" fallita:"
 20     echo "File \"$0\", riga $numriga"
 21     exit $E_ASSERT_FALLITA
 22   # else
 23   #   return
 24   #   e continua l'esecuzione dello script.
 25   fi
 26 }
 27
 28
 29 a=5
 30 b=4
 31 condizione="$a -lt $b"    #  Messaggio d'errore ed uscita dallo
script.
 32                          #  Provate ad impostare "condizione" con
 33                          #+ qualcos'altro, e vedete cosa succede.
 34
 35 assert "$condizione" $LINENO
 36 # La parte restante dello script verrà eseguita solo se "assert" non
fallisce.
 37
 38
 39 # Alcuni comandi.
 40 # ...
 41 echo "Questo enunciato viene visualizzato solo se \"assert\" non
fallisce."
 42 # ...
 43 # Alcuni altri comandi.
 44
 45 exit 0
```

5. eseguire una trap di exit.

   Il comando **exit**, , in uno script, lancia il segnale 0 che termina il processo, cioè, lo script stesso. [2] È spesso utile eseguire una trap di **exit**, per esempio, per forzare la "visualizzazione" delle variabili. trap deve essere il primo comando dello script.

Trap dei segnali

trap

Specifica un'azione che deve essere eseguita alla ricezione di un segnale; è utile anche per il debugging.

Un *segnale* è semplicemente un messaggio inviato ad un processo, o dal kernel o da un altro processo, che gli comunica di eseguire un'azione specifica (solitamente di terminare). Per esempio, la pressione di **Control-C** invia un interrupt utente, il segnale INT, al programma in esecuzione.

```
trap '' 2
# Ignora l'interrupt 2 (Control-C), senza alcuna azione specificata.

trap 'echo "Control-C disabilitato."' 2
# Messaggio visualizzato quando si digita Control-C.
```

Esempio 30-5. Trap di exit

```
#!/bin/bash
# Andare a caccia di variabili con trap.

trap 'echo Elenco Variabili --- a = $a  b = $b' EXIT
#  EXIT è il nome del segnale generato all'uscita dallo script.

#  Il comando specificato in "trap" non viene eseguito finché
#+ non è stato inviato il segnale appropriato.

echo "Questa visualizzazione viene eseguita prima di \"trap\" --"
echo "nonostante lo script veda prima \"trap\"."
echo

a=39

b=36

exit 0
#  Notate che anche se si commenta il comando 'exit' questo non fa
#+ alcuna differenza, poiché lo script esce in ogni caso dopo
#+ l'esecuzione dei comandi.
```

Esempio 30-6. Pulizia dopo un Control-C

```
#!/bin/bash
#  logon.sh: Un rapido e rudimentale script per verificare se si
#+ è ancora collegati.


TRUE=1
FILELOG=/var/log/messages
#  Fate attenzione che $FILELOG deve avere i permessi di lettura
#+ (da root, chmod 644 /var/log/messages).
FILETEMP=temp.$$
#  Crea un file temporaneo con un nome "unico", usando l'id di
#+ processo dello script.
PAROLACHIAVE=address
#  A collegamento avvenuto, la riga "remote IP address xxx.xxx.xxx.xxx"
#                                  viene accodata in /var/log/messages.
COLLEGATO=22
INTERRUPT_UTENTE=13
CONTROLLA_RIGHE=100
#  Numero di righe del file di log da controllare.
```

```
 20
 21 trap 'rm -f $FILETEMP; exit $INTERRUPT_UTENTE'; TERM INT
 22 #  Cancella il file temporaneo se lo script viene interrotto con un control-
c.
 23
 24 echo
 25
 26 while [ $TRUE ]  # Ciclo infinito.
 27 do
 28   tail -$CONTROLLA_RIGHE $FILELOG> $FILETEMP
 29   #  Salva le ultime 100 righe del file di log di sistema nel file
 30   #+ temporaneo. Necessario, dal momento che i kernel più
 31   #+ recenti generano molti messaggi di log durante la fase di avvio.
 32   ricerca=`grep $PAROLACHIAVE $FILETEMP`
 33   #  Verifica la presenza della frase "IP address",
 34   #+ che indica che il collegamento è riuscito.
 35
 36   if [ ! -z "$ricerca" ] #  Sono necessari gli apici per la possibile
 37                          #+ presenza di spazi.
 38   then
 39      echo "Collegato"
 40      rm -f $FILETEMP     #  Cancella il file temporaneo.
 41      exit $COLLEGATO
 42   else
 43       echo -n "."        #  L'opzione -n di echo sopprime il ritorno a capo,
 44                          #+ così si ottengono righe continue di punti.
 45   fi
 46
 47   sleep 1
 48 done
 49
 50
 51 #  Nota: se sostituite la variabile PAROLACHIAVE con "Exit",
 52 #+ potete usare questo script per segnalare, mentre si è collegati,
 53 #+ uno scollegamento inaspettato.
 54
 55 #  Esercizio: Modificate lo script per ottenere quanto suggerito nella
 56 #             nota precedente, rendendolo anche più elegante.
 57
 58 exit 0
 59
 60
 61 # Nick Drage ha suggerito un metodo alternativo:
 62
 63 while true
 64   do ifconfig ppp0 | grep UP 1> /dev/null && echo "connesso" && exit 0
 65   echo -n "."   # Visualizza dei punti (.....) finché si è connessi.
 66   sleep 2
 67 done
 68
 69 # Problema: Può non bastare premere Control-C per terminare il processo.
 70 #+          (La visualizzazione dei punti potrebbe continuare.)
 71 # Esercizio: Risolvetelo.
 72
 73
 74
 75 # Stephane Chazelas ha un'altra alternativa ancora:
 76
 77 INTERVALLO=1
 78
 79 while ! tail -1 "$FILELOG" | grep -q "$PAROLACHIAVE"
 80 do echo -n .
```

```
81    sleep $INTERVALLO
82 done
83 echo "Connesso"
84
85 # Esercizio: Discutete i punti di forza e i punti deboli
86 #            di ognuno di questi differenti approcci.
```

Fornendo `DEBUG` come argomento a **trap**, viene eseguita l'azione specificata dopo ogni comando presente nello script. Questo consente, per esempio, il tracciamento delle variabili.

Esempio 30-7. Tracciare una variabile

```
 1 #!/bin/bash
 2
 3 trap 'echo "TRACCIA-VARIABILE> \$variabile = \"$variabile\""' DEBUG
 4 # Visualizza il valore di $variabile dopo l'esecuzione di ogni comando.
 5
 6 variabile=29;
 7
 8 echo "La \"\$variabile\" è stata inizializzata a $variabile."
 9
10 let "variabile *= 3"
11 echo "\"\$variabile\" è stata moltiplicata per 3."
12
13 #  Il costrutto "trap 'comandi' DEBUG" diventa molto utile
14 #+ nel contesto di uno script complesso, dove collocare molti
15 #+ enunciati "echo $variabile" si rivela goffo oltre che una perdita
16 #+ di tempo.
17
18 # Grazie, Stephane Chazelas per la puntualizzazione.
19
20 exit 0
```

Naturalmente, il comando **trap** viene impiegato per altri scopi oltre a quello per il debugging.

Esempio 30-8. Esecuzione di processi multipli (su una postazione SMP)

```
 1 #!/bin/bash
 2 # multiple-processes.sh: Esegue processi multipli su una macchina SMP.
 3
 4 # Script di Vernia Damiano.
 5 # Usato con il suo permesso.
 6
 7 #  Lo script deve essere richiamato con almeno un parametro numerico
 8 #+ (numero dei processi simultanei).
 9 #  Tutti gli altri parametri sono passati ai processi in esecuzione.
10
11
12 INDICE=8        # Numero totale di processi da mettere in esecuzione
13 TEMPO=5         # Tempo massimo d'attesa per processo
14 E_NOARG=65      # Nessun argomento(i) passato allo script.
15
16 if [ $# -eq 0 ] # Controlla la presenza di almeno un argomento.
17 then
18   echo "Utilizzo: `basename $0` numero_dei_processi [parametri passati]"
19   exit $E_NOARG
20 fi
21
22 NUMPROC=$1              # Numero dei processi simultanei
23 shift
```

```
 24 PARAMETRI=( "$@" )      # Parametri di ogni processo
 25
 26 function avvia() {
 27     local temp
 28     local index
 29     temp=$RANDOM
 30     index=$1
 31     shift
 32     let "temp %= $TEMPO"
 33     let "temp += 1"
 34     echo "Inizia $index Tempo:$temp" "$@"
 35     sleep ${temp}
 36     echo "Termina $index"
 37     kill -s SIGRTMIN $$
 38 }
 39
 40 function parti() {
 41     if [ $INDICE -gt 0 ] ; then
 42             avvia $INDICE "${PARAMETRI[@]}" &
 43             let "INDICE--"
 44     else
 45             trap : SIGRTMIN
 46     fi
 47 }
 48
 49 trap parti SIGRTMIN
 50
 51 while [ "$NUMPROC" -gt 0 ]; do
 52     parti;
 53     let "NUMPROC--"
 54 done
 55
 56 wait
 57 trap - SIGRTMIN
 58
 59 exit $?
 60
 61 : << COMMENTO_DELL'AUTORE_DELLO_SCRIPT
 62 Avevo la necessità di eseguire un programma, con determinate opzioni, su un
 63 numero diverso di file, utilizzando una macchina SMP. Ho pensato, quindi, di
 64 mantenere in esecuzione un numero specifico di processi e farne iniziare uno
 65 nuovo ogni volta . . . che uno di quest'ultimi terminava.
 66
 67 L'istruzione "wait" non è d'aiuto, poichè attende sia per un dato processo
 68 sia per *tutti* i processi in esecuzione sullo sfondo (background). Ho
scritto,
 69 di conseguenza, questo script che è in grado di svolgere questo compito,
 70 usando l'istruzione "trap".
 71   --Vernia Damiano
 72 COMMENTO_DELL'AUTORE_DELLO_SCRIPT
```

`trap '' SEGNALE` (due apostrofi adiacenti) disabilita SEGNALE nella parte restante dello script. **`trap SEGNALE`** ripristina nuovamente la funzionalità di SEGNALE. È utile per proteggere una parte critica dello script da un interrupt indesiderato.

```
 1        trap '' 2  #  Il segnale 2 è Control-C, che ora è disabilitato.
 2        comando
 3        comando
 4        comando
 5        trap 2     # Riabilita Control-C
 6
```

La versione 3 di Bash ha aggiunto le variabili speciali seguenti ad uso di chi deve eseguire il debugging.

1. $BASH_ARGC
2. $BASH_ARGV
3. $BASH_COMMAND
4. $BASH_EXECUTION_STRING
5. $BASH_LINENO
6. $BASH_SOURCE
7. $BASH_SUBSHELL

Note

[1] Il Bash debugger di Rocky Bernstein colma, in parte, questa lacuna.

[2] Convenzionalmente, il *`segnale 0`* è assegnato a exit.

Capitolo 31. Opzioni

Le opzioni sono impostazioni che modificano il comportamento della shell e/o dello script.

Il comando set abilita le opzioni in uno script. Nel punto dello script da cui si vuole che le opzioni abbiano effetto, si inserisce **set -o nome-opzione** o, in forma abbreviata, **set -abbrev-opzione**. Le due forme si equivalgono.

```
1       #!/bin/bash
2
3       set -o verbose
4       # Visualizza tutti i comandi prima della loro esecuzione.
5
```

```
1       #!/bin/bash
2
3       set -v
4       # Identico effetto del precedente.
5
```

 Per *disabilitare* un'opzione in uno script, si usa **set +o nome-opzione** o **set +abbrev-opzione**.

```
1       #!/bin/bash
2
3       set -o verbose
4       # Abilitata la visualizzazione dei comandi.
5       comando
6       ...
7       comando
8
9       set +o verbose
10      # Visualizzazione dei comandi disabilitata.
11      comando
12      # Non visualizzato.
13
14
```

```
15      set -v
16      # Visualizzazione dei comandi abilitata.
17      comando
18      ...
19      comando
20
21      set +v
22      # Visualizzazione dei comandi disabilitata.
23      comando
24
25      exit 0
26
```

Un metodo alternativo per abilitare le opzioni in uno script consiste nello specificarle immediatamente dopo l'intestazione *#!*.

```
1       #!/bin/bash -x
2       #
3       # Corpo dello script.
4
```

È anche possibile abilitare le opzioni per uno script da riga di comando. Alcune di queste, che non si riesce ad impostare con **set**, vengono rese disponibili per questa via. Tra di esse *-i*, che forza l'esecuzione interattiva dello script.

```
bash -v nome-script

bash -o verbose nome-script
```

Quello che segue è un elenco di alcune delle opzioni più utili. Possono essere specificate sia in forma abbreviata (precedute da un trattino singolo) che con il loro nome completo (precedute da un *doppio* trattino o da `-o`).

Tabella 31-1. Opzioni bash

Abbreviazione	Nome	Effetto
`-C`	noclobber	Evita la sovrascrittura dei file a seguito di una redirezione (può essere annullato con >|)
`-D`	(nessuno)	Elenca le stringhe tra doppi apici precedute da $, ma non esegue i comandi nello script
`-a`	allexport	Esporta tutte le variabili definite
`-b`	notify	Notifica la terminazione dei job in esecuzione in background (non molto utile in uno script)
`-c ...`	(nessuno)	Legge i comandi da **...**
`-e`	errexit	Lo script abortisce al primo errore, quando un comando termina con un exit status diverso da zero (ad eccezione dei cicli until o while, verifiche if, costrutti lista)
`-f`	noglob	Disabilita l'espansione dei nomi dei file (globbing)
`-i`	interactive	Lo script viene eseguito in modalità *interattiva*
`-n`	noexec	Legge i comandi dello script, ma non li esegue (controllo di sintassi)

Abbreviazione	Nome	Effetto
`-o Nome-Opzione`	(nessuno)	Invoca l'opzione *Nome-Opzione*
`-o posix`	POSIX	Modifica il comportamento di Bash, o dello script da eseguire, per conformarlo allo standard POSIX .
`-p`	privileged	Lo script viene eseguito con il bit "suid" impostato (attenzione!)
`-r`	restricted	Lo script viene eseguito in modalità *ristretta* (vedi Capitolo 21).
`-s`	stdin	Legge i comandi dallo `stdin`
`-t`	(nessuno)	Esce dopo il primo comando
`-u`	nounset	Il tentativo di usare una variabile non definita provoca un messaggio d'errore e l'uscita forzata dallo script
`-v`	verbose	Visualizza ogni comando allo `stdout` prima della sua esecuzione
`-x`	xtrace	Simile a `-v`, ma espande i comandi
`-`	(nessuno)	Indicatore di fine delle opzioni. Tutti gli altri argomenti sono considerati parametri posizionali.
`--`	(nessuno)	Annulla i parametri posizionali. Se vengono forniti degli argomenti (`-- arg1 arg2`), i parametri posizionali vengono impostati agli argomenti.

Capitolo 32. Precauzioni

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!
Puccini

Non usare, per i nomi delle variabili, parole o caratteri riservati.

```
 1 case=valore0       # Causa problemi.
 2 23skidoo=valore1   # Ancora problemi.
 3 #  I nomi di variabili che iniziano con una cifra sono riservati alla shell.
 4 #  Sostituite con  _23skidoo=valore1. I nomi che iniziano con un
 5 #+ underscore (trattino di sottolineatura) vanno bene.
 6
 7 # Tuttavia...      usare il solo underscore non funziona.
 8 _=25
 9 echo $_            #  $_ è la variabile speciale impostata
10                    #+ all'ultimo argomento dell'ultimo comando.
11
12 xyz((!*=valore2    # Provoca seri problemi.
```

Non usare il trattino o altri caratteri riservati nel nome di una variabile.

```
1 var-1=23 # Usate invece 'var_1'.
```

Non usare lo stesso nome per una variabile e per una funzione. Ciò rende lo script difficile da capire.

```
 1 fa_qualcosa ()
 2 {
 3   echo "Questa funzione fa qualcosa con \"$1\"."
 4 }
 5
 6 fa_qualcosa=fa_qualcosa
 7
 8 fa_qualcosa fa_qualcosa
 9
10 # Tutto questo è consentito, ma estremamente sconcertante.
```

Non usare impropriamente gli spazi. A differenza di altri linguaggi di programmazione, Bash è piuttosto pignola con gli spazi.

```
 1 var1 = 23   # corretto 'var1=23'.
 2 # Nella riga precedente, Bash cerca di eseguire il comando "var1"
 3 # con gli argomenti "=" e "23".
 4
 5 let c = $a - $b   # corretto 'let c=$a-$b' o 'let "c = $a - $b"'.
 6
 7 if [ $a -le 5]    # corretto if [ $a -le 5 ]   .
 8 # if [ "$a" -le 5 ]   ancora meglio.
 9 # [[ $a -le 5 ]] anche così.
```

Non dare per scontato che le variabili non inizializzate (variabili a cui non è ancora stato assegnato un valore) valgano "zero". Una variabile non inizializzata ha valore "nullo", "non" zero.

```
 1 #!/bin/bash
 2
 3 echo "var_non_inizializzata = $var_non_inizializzata"
 4 # var_non_inizializzata =
```

Non confondere = e *-eq* nelle verifiche. Bisogna ricordarsi che = serve per il confronto tra variabili letterali mentre *-eq* per quello tra interi.

```
 1 if [ "$a" = 273 ]      # $a è un intero o una stringa?
 2 if [ "$a" -eq 273 ]    # $a è un intero.
 3
 4 # Talvolta è possibile scambiare -eq con = senza alcuna conseguenza.
 5 # Tuttavia...
 6
 7
 8 a=273.0   # Non è un intero.
 9
10 if [ "$a" = 273 ]
11 then
12   echo "Il confronto ha funzionato."
13 else
14   echo "Il confronto non ha funzionato."
15 fi    # Il confronto non ha funzionato.
16
17 # Stessa cosa con   a=" 273"  e a="0273".
18
19
20 # Allo stesso modo, si hanno problemi ad usare "-eq" con valori non interi.
21
22 if [ "$a" -eq 273.0 ]
23 then
```

```
 24   echo "a = $a'
 25 fi  # Esce con un messaggio d'errore.
 26 # test.sh: [: 273.0: integer expression expected
```

Non usare in modo scorretto gli operatori per il confronto di stringhe.

Esempio 32-1. I confronti numerici e quelli di stringhe non si equivalgono

```
  1 #!/bin/bash
  2 # bad-op.sh: Tentativo di usare il confronto di stringhe con gli interi.
  3
  4 echo
  5 numero=1
  6
  7 # Il "ciclo while" seguente contiene due errori:
  8 #+ uno vistoso, l'altro subdolo.
  9
 10 while [ "$numero" < 5 ]    # Errato! Dovrebbe essere:  while [ "$numero" -lt
5 ]
 11 do
 12   echo -n "$numero "
 13   let "numero += 1"
 14 done
 15 #  La sua esecuzione provoca il messaggio d'errore:
 16 #+ bad-op.sh: line 10: 5: No such file or directory
 17 #  All'interno delle parentesi quadre singole si deve applicare il quoting
a"<"
 18 #+ e, anche così, è sbagliato usarlo per confrontare gli interi.
 19
 20
 21 echo "----------------------"
 22
 23
 24 while [ "$numero" \< 5 ]    #  1 2 3 4
 25 do                         #
 26   echo -n "$numero "        #  Questo *sembra funzionare, ma . . .
 27   let "numero += 1"         #+ in realtà esegue un confronto ASCII,
 28 done                       #+ invece di uno numerico.
 29
 30 echo; echo "----------------------"
 31
 32 # Questo può provocare dei problemi. Ad esempio:
 33
 34 minore=5
 35 maggiore=105
 36
 37 if [ "$maggiore" \< "$minore" ]
 38 then
 39   echo "$maggiore è minore di $minore"
 40 fi                           # 105 è minore di 5
 41 #  Infatti, "105" è veramente minore di "5"
 42 #+ in un confronto di stringhe (ordine ASCII).
 43
 44 echo
 45
 46 exit 0
```

Talvolta è necessario il quoting (apici doppi) per le variabili che si trovano all'interno del costrutto di "verifica" parentesi quadre ([]). Non farne uso può causare un comportamento inaspettato. Vedi Esempio 7-6, Esempio 16-5 e Esempio 9-6.

Comandi inseriti in uno script possono fallire l'esecuzione se il proprietario dello script non ha, per quei comandi, i permessi d'esecuzione. Se un utente non può invocare un comando al prompt di shell, il fatto di inserirlo in uno script non cambia la situazione. Si provi a cambiare gli attributi dei comandi in questione, magari impostando il bit suid (come root, naturalmente).

Cercare di usare il **-** come operatore di redirezione (che non è) di solito provoca spiacevoli sorprese.

```
 1 comando1 2> - | comando2  #  Il tentativo di redirigere l'output
 2                          #+ d'errore di comando 1 con una pipe...
 3                          #    ...non funziona.
 4
 5 comando1 2>& - | comando2 # Altrettanto inutile.
 6
 7 Grazie, S.C.
```

Usare le funzionalità di Bash versione 2+ può provocare l'uscita dal programma con un messaggio d'errore. Le macchine Linux più datate potrebbero avere, come installazione predefinita, la versione Bash 1.XX.

```
 1 #!/bin/bash
 2
 3 versione_minima=2
 4 # Dal momento che Chet Ramey sta costantemente aggiungendo funzionalità a
Bash,
 5 # si può impostare $versione_minima a 2.XX, o ad altro valore appropriato.
 6 E_ERR_VERSIONE=80
 7
 8 if [ "$BASH_VERSION" \< "$versione_minima" ]
 9 then
10   echo "Questo script funziona solo con Bash, versione"
11   echo "$versione_minima o superiore."
12   echo "Se ne consiglia caldamente l'aggiornamento."
13   exit $E_ERR_VERSIONE
14 fi
15
16 ...
```

Usare le funzionalità specifiche di Bash in uno script di shell Bourne (**#!/bin/sh**) su una macchina non Linux può provocare un comportamento inatteso. Un sistema Linux di solito esegue l'alias di **sh** a **bash**, ma questo non è necessariamente vero per una generica macchina UNIX.

Usare funzionalità non documentate in Bash può rivelarsi una pratica pericolosa. Nelle versioni precedenti di questo libro erano presenti diversi script che si basavano su una "funzionalità" che, sebbene il valore massimo consentito per exit o return fosse 255, permetteva agli interi *negativi* di superare tale limite. Purtroppo, con la versione 2.05b e successive, tale scappatoia è scomparsa. Vedi Esempio 23-8.

Uno script con i caratteri di a capo di tipo DOS (`\r\n`) fallisce l'esecuzione poiché **#!/bin/bash\r\n** non viene riconosciuto, *non* è la stessa cosa dell'atteso **#!/bin/bash\n**. La correzione consiste nel convertire tali caratteri nei corrispondenti UNIX.

```
 1 #!/bin/bash
 2
 3 echo "Si parte"
 4
```

```
 5 unix2dos $0    # lo script si trasforma nel formato DOS.
 6 chmod 755 $0   # Viene ripristinato il permesso di esecuzione.
 7                # Il comando 'unix2dos' elimina i permessi di escuzione.
 8
 9 ./$0           # Lo script tenta la riesecuzione.
10                # Come file DOS non può più funzionare.
11
12 echo "Fine"
13
14 exit 0
```

Uno script di shell che inizia con **#!/bin/sh** non funziona in modalità di piena compatibilità Bash. Alcune funzioni specifiche di Bash potrebbero non essere abilitate. Gli script che necessitano di un accesso completo a tali estensioni devono iniziare con **#!/bin/bash**.

Mettere degli spazi davanti alla stringa limite di chiusura di un here document provoca un comportamento inatteso dello script.

Uno script non può esportare (**export**) le variabili in senso contrario né verso il suo processo genitore, la shell, né verso l'ambiente. Proprio come insegna la biologia, un figlio può ereditare da un genitore, ma non viceversa.

```
 1 QUELLO_CHE_VUOI=/home/bozo
 2 export QUELLO_CHE_VUOI
 3 exit 0
```

```
bash$ echo $QUELLO_CHE_VUOI

bash$
```

È sicuro, al prompt dei comandi, $QUELLO_CHE_VUOI rimane non impostata.

Impostare e manipolare variabili all'interno di una subshell e cercare, successivamente, di usare quelle stesse variabili al di fuori del loro ambito, provocherà una spiacevole sorpresa.

Esempio 32-2. I trabocchetti di una subshell

```
 1 #!/bin/bash
 2 # Le insidie delle variabili di una subshell.
 3
 4 variabile_esterna=esterna
 5 echo
 6 echo "variabile esterna = $variabile_esterna"
 7 echo
 8
 9 (
10 # Inizio della subshell
11
12 echo "variabile esterna nella subshell = $variabile_esterna"
13 variabile_interna=interna  # Impostata
14 echo "variabile interna nella subshell = $variabile_interna"
15 variabile_esterna=interna  # Il valore risulterà cambiato a livello globale?
16 echo "variabile esterna nella subshell = $variabile_esterna"
17
18 # Fine della subshell
19 )
20
```

```
21 echo
22 echo "variabile interna al di fuori della subshell = $variabile_interna"
23               # Non impostata.
24 echo "variabile esterna al di fuori della subshell = $variabile_esterna"
25               # Immutata.
26 echo
27
28 exit 0
```

Collegare con una pipe l'output di **echo** a read può produrre risultati inattesi. In un tale scenario, **read** si comporta come se fosse in esecuzione all'interno di una subshell. Si usi invece il comando set (come in Esempio 11-15).

Esempio 32-3. Concatenare con una pipe l'output di echo a read

```
 1 #!/bin/bash
 2 #  badread.sh:
 3 #  Tentativo di usare 'echo e 'read'
 4 #+ per l'assegnazione non interattiva di variabili.
 5
 6 a=aaa
 7 b=bbb
 8 c=ccc
 9
10 echo "uno due tre" | read a b c
11 # Cerca di riassegnare a, b e c.
12
13 echo
14 echo "a = $a"  # a = aaa
15 echo "b = $b"  # b = bbb
16 echo "c = $c"  # c = ccc
17 # Riassegnazione fallita.
18
19 # ------------------------------
20
21 # Proviamo la seguente alternativa.
22
23 var=`echo "uno due tre"`
24 set -- $var
25 a=$1; b=$2; c=$3
26
27 echo "-------"
28 echo "a = $a"  # a = uno
29 echo "b = $b"  # b = due
30 echo "c = $c"  # c = tre
31 # Riassegnazione riuscita.
32
33 # ------------------------------
34
35 #  Notate inoltre che echo con 'read' funziona all'interno di una subshell.
36 #  Tuttavia, il valore della variabile cambia *solo* in quell'ambito.
37
38 a=aaa          # Ripartiamo da capo.
39 b=bbb
40 c=ccc
41
42 echo; echo
43 echo "uno due tre" | ( read a b c;
44 echo "nella subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
45 # a = uno
```

```
46 # b = due
47 # c = tre
48 echo "----------------------"
49 echo "Fuori dalla subshell: "
50 echo "a = $a"  # a = aaa
51 echo "b = $b"  # b = bbb
52 echo "c = $c"  # c = ccc
53 echo
54
55 exit 0
```

Infatti, come fa notare Anthony Richardson, usare la pipe con *qualsiasi* ciclo può provocare un simile problema.

```
 1 # Problemi nell'uso di una pipe con un ciclo.
 2 #  Esempio di Anthony Richardson.
 3 #+ con appendice di Wilbert Berendsen.
 4
 5
 6 trovato=falso
 7 find $HOME -type f -atime +30 -size 100k |
 8 while true
 9 do
10    read f
11    echo "$f supera i 100KB e non è stato usato da più di 30 giorni"
12    echo "Considerate la possibilità di spostarlo in un archivio."
13    trovato=vero
14    # -----------------------------------------
15    echo "Livello della subshell = $BASH_SUBSHELL"
16    # Livello della subshell = 1
17    # Si, siete all'interno di una subshell.
18    # -----------------------------------------
19 done
20
21 #  In questo caso trovato sarà sempre falso perchè
22 #+ è stato impostato all'interno di una subshell
23 if [ $trovato = falso ]
24 then
25    echo "Nessun file da archiviare."
26 fi
27
28 # ====================Ora nel modo corretto:=======================
29
30 trovato=falso
31 for f in $(find $HOME -type f -atime +30 -size 100k)  # Nessuna pipe.
32 do
33    echo "$f supera i 100KB e non è stato usato da più di 30 giorni"
34    echo "Considerate la possibilità di spostarlo in un archivio."
35    trovato=vero
36 done
37
38 if [ $trovato = falso ]
39 then
40    echo "Nessun file da archiviare."
41 fi
42
43 # ====================Ed ecco un'altra alternativa=================
44
45 #  Inserite la parte dello script che legge le variabili all'interno del
46 #+ blocco di codice, in modo che condividano la stessa subshell.
47 #  Grazie, W.B.
```

```
48
49 find $HOME -type f -atime +30 -size 100k | {
50      trovato=false
51      while read f
52      do
53        echo "$f supera i 100KB e non è stato usato da più di 30 giorni"
54        echo "Considerate la possibilità di spostarlo in un archivio."
55        trovato=true
56      done
57
58      if ! $trovato
59      then
60        echo "Nessun file da archiviare."
61      fi
```

Un problema simile si verifica quando si cerca di scrivere lo `stdout` di **tail -f** collegato con una pipe a grep.

```
 1 tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
 2 # Nel file "error.log" non ci sarà scritto niente.
```

--

È rischioso, negli script, l'uso di comandi che hanno il bit "suid" impostato, perché questo può compromettere la sicurezza del sistema. [1]

L'uso degli script di shell per la programmazione CGI potrebbe rivelarsi problematica. Le variabili degli script di shell non sono "tipizzate" e questo fatto può causare un comportamento indesiderato per quanto concerne CGI. Inoltre, è difficile proteggere dal "cracking" gli script di shell.

Bash non gestisce correttamente la stringa doppia barra (//).

Gli script Bash, scritti per i sistemi Linux o BSD, possono aver bisogno di correzioni per consentire la loro esecuzione su macchine UNIX commerciali. Questi script, infatti, fanno spesso uso di comandi e filtri GNU che hanno funzionalità superiori ai loro corrispettivi generici UNIX. Questo è particolarmente vero per le utility di elaborazione di testo come tr.

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.

A.J. Lamb and H.W. Petrie

Note

[1] L'impostazione del bit *suid* dello script stesso non ha alcun effetto.

Capitolo 33. Stile dello scripting

Ci si abitui a scrivere gli script di shell in modo strutturato e sistematico. Anche "al volo" e "scritti sul retro di una busta", gli script trarranno beneficio se si dedicano pochi minuti a pianificare ed organizzare le idee prima di sedersi a codificarle.

Ecco di seguito poche linee guida per lo stile. Non devono essere intese come *Regole di stile ufficiali per lo scripting di shell.*.

33.1. Regole di stile non ufficiali per lo scripting di shell

- Si commenti il codice. I commenti rendono più facile agli altri capirlo (e apprezzarlo) e più semplice la sua manutenzione.

```
  1 PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
  2 #  Aveva perfettamente senso quando, l'anno scorso, l'avevate
scritto, ma
  3 #+ adesso è un mistero totale.
  4 # (Da Antek Sawicki's "pw.sh" script.)
```

- Si aggiungano intestazioni descrittive agli script e alle funzioni.

```
  1 #!/bin/bash
  2
  3 #********************************************#
  4 #                   xyz.sh                   #
  5 #           scritto da Bozo Bozeman          #
  6 #               05 luglio 2001               #
  7 #                                            #
  8 #      Cancellazione dei file di progetto.   #
  9 #********************************************#
 10
 11 E_ERRDIR=65                      # Directory inesistente.
 12 dirprogetti=/home/bozo/projects   # Directory da cancellare.
 13
 14 # ------------------------------------------------------------------
-- #
 15 # cancella_filep ()
#
 16 # Cancella tutti i file della directory specificata.
#
 17 # Parametro: $directory_indicata
#
 18 # Restituisce: 0 in caso di successo, $E_ERRDIR se qualcosa va
storto. #
 19 # ------------------------------------------------------------------
-- #
 20 cancella_filep ()
 21 {
 22   if [ ! -d "$1" ]  # Verifica l'esistenza della directory indicata.
 23   then
 24      echo "$1 non è una directory."
 25      return $E_ERRDIR
```

```
 26    fi
 27
 28   rm -f "$1"/*
 29   return 0   # Successo.
 30 }
 31
 32 cancella_filep $dirprogetti
 33
 34 exit 0
```

- Ci si accerti di aver posto *#!/bin/bash* all'inizio della prima riga dello script, prima di qualsiasi commento.
- Si eviti di usare, per i nomi delle costanti letterali, dei "magic number", [1] cioè, costanti "codificate" . Si utilizzino invece nomi di variabile significativi. Ciò renderà gli script più facili da capire e consentirà di effettuare le modifiche e gli aggiornamenti senza il pericolo che l'applicazione non funzioni più correttamente.

```
  1 if [ -f /var/log/messages ]
  2 then
  3   ...
  4 fi
  5 #  L'anno successivo decidete di cambiare lo script per
  6 #+ verificare /var/log/syslog.
  7 #  È necessario modificare manualmente lo script, un'occorrenza
  8 #+ alla volta, e sperare che tutto funzioni a dovere.
  9
 10 # Un modo migliore:
 11 FILELOG=/var/log/messages  # Basterà cambiare solo questa riga.
 12 if [ -f "$FILELOG" ]
 13 then
 14   ...
 15 fi
```

- Si scelgano nomi descrittivi per le variabili e le funzioni.

```
  1 ef=`ls -al $nomedir`                  #  Criptico.
  2 elenco_file=`ls -al $nomedir`         #  Meglio.
  3
  4
  5 VALMAX=10                             #  I nomi delle costanti in
  6                                       #+ lettere maiuscole.
  7 while [ "$indice" -le "$VALMAX" ]
  8 ...
  9
 10
 11 E_NONTROVATO=75                       #  Costanti dei codici d'errore
 12                                       #+ in maiuscolo,
 13                                       #  e con i nomi che iniziano
con "E_".
 14 if [ ! -e "$nomefile" ]
 15 then
 16   echo "Il file $nomefile non è stato trovato."
 17   exit $E_NONTROVATO
 18 fi
 19
 20
 21 MAIL_DIRECTORY=/var/spool/mail/bozo  #  Lettere maiuscole per le
variabili
 22                                      #+ d'ambiente.
```

```
 23 export MAIL_DIRECTORY
 24
 25
 26 LeggiRisposta ()                    #  Iniziali maiuscole per i
nomi di
 27                                     #+ funzione.
 28 {
 29   prompt=$1
 30   echo -n $prompt
 31   read risposta
 32   return $risposta
 33 }
 34
 35 LeggiRisposta "Qual'è il tuo numero preferito? "
 36 numero_preferito=$?
 37 echo $numero_preferito
 38
 39
 40 _variabileutente=23                # Consentito, ma non
raccomandato.
 41 #  È preferibile che i nomi delle variabili definite dall'utente non
inizino
 42 #+ con un underscore.
 43 #  Meglio lasciarlo per le variabili di sistema.
```

- Si faccia uso dei codici di uscita in modo sistematico e significativo.

```
  1 E_ERR_ARG=65
  2 ...
  3 ...
  4 exit $E_ERR_ARG
```

- Vedi anche Appendice D.
- *Ender* suggerisce di usare, per gli script di shell, i codici di exit elencati in `/usr/include/sysexits.h` , sebbene questi si riferiscano alla programmazione in C e C++.
- Nell'invocazione di uno script si usino le opzioni standard. *Ender* propone la serie seguente.

```
  1 -a      Tutto (all): informazioni complete (comprese quelle
  2         riguardanti i file nascosti).
  3 -b      Breve: versione abbreviata, solitamente per altri script.
  4 -c      Copia, concatena, ecc.
  5 -d      Giornaliero (daily): informazioni sull'intera giornata, non
solo quelle
  6         di uno/a specifico/a utente/istanza.
  7 -e      Esteso/Elaborato: (spesso non comprende informazioni sui
file nascosti).
  8 -h      Aiuto (help): dettagli sull'uso w/desc, info aggiuntive,
discussione.
  9         Vedi anche -V.
 10 -l      Registra l'output dello script.
 11 -m      Manuale: visualizza la pagina di manuale di un comando di
base.
 12 -n      Numeri: solo dati numerici.
 13 -r      Ricorsivo: tutti i file di una directory (e/o tutte le sub-
directory).
 14 -s      Impostazioni (setup) & Gestione File: file di configurazione
 15         dello script.
 16 -u      Utilizzo: elenco delle opzioni d'esecuzione dello script.
```

```
 17 -v      Dettaglio (verbose): informazioni dettagliate, più o meno
formattate.
 18 -V      Versione / Licenza / Copy(right|left) / Contributi (anche
email).
```

- Vedi anche Appendice F.
- Si suddividano gli script complessi in moduli più semplici. Si faccia uso delle funzioni ogni qual volta se ne presenti l'occasione. Vedi Esempio 35-4.
- Non si usi un costrutto complesso dove uno più semplice è sufficiente.

```
 1 COMANDO if [ $? -eq 0 ]
 2 ...
 3 # Ridondante e non intuitivo.
 4
 5 if COMANDO
 6 ...
 7 # Più conciso (anche se, forse, non altrettanto leggibile).
```

... reading the UNIX source code to the Bourne shell (/bin/sh). I was shocked at how much simple algorithms could be made cryptic, and therefore useless, by a poor choice of code style. I asked myself, "Could someone be proud of this code?"

Landon Noll

Note

[1] In questo contesto, il termine "magic number" ha un significato completamente diverso dal magic number usato per designare i tipi di file.

Capitolo 34. Miscellanea

Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help.

Tom Duff

Sommario

34.1. Shell e script interattivi e non

Una shell *interattiva* legge i comandi dall'input dell'utente, immessi da una `tty`. Una tale shell, in modo predefinito, legge i file di avvio in fase di attivazione, visualizza un prompt e abilita il controllo dei job, tra le altre cose. L'utente può *interagire* con la shell.

Una shell che esegue uno script è sempre una shell non interattiva. Tuttavia, lo script può ancora accedere alla sua `tty`. È anche possibile simulare, nello script, una shell interattiva.

```
 1 #!/bin/bash
 2 MIO_PROMPT='$ '
 3 while :
 4 do
 5   echo -n "$MIO_PROMPT"
 6   read riga
 7   eval "$riga"
 8   done
 9
10 exit 0
11
12 #  Questo script d'esempio e gran parte della spiegazione precedente
13 #+ sono stati forniti da Stephane Chazelas (grazie ancora).
```

Si considera come *interattivo* quello script che richiede l'input dall'utente, di solito per mezzo di enunciati read (vedi Esempio 11-2). La "realtà" , a dire il vero, è un po' meno semplice di così. Per il momento si assume che uno script interattivo sia quello connesso ad una tty, uno script che un utente ha invocato da console o da xterm.

Gli script init e di avvio sono, per forza di cose, non interattivi, perché devono essere eseguiti senza l'intervento umano. Allo stesso modo, non sono interattivi gli script che svolgono attività d'amministrazione e di manutenzione del sistema. Compiti invariabili e ripetitivi richiedono di essere svolti automaticamente per mezzo di script non interattivi.

Gli script non interattivi possono essere eseguiti in background, a differenza di quelli interattivi che si bloccano in attesa di un input che non arriverà mai. Questa difficoltà può essere gestita con uno script **expect** o con l'inserimento di un here document che sono in grado di fornire l'input allo script interattivo in esecuzione in background. Nel caso più semplice, redirigendo un file per fornire l'input ad un enunciato **read** (**read variabile <file**). Questi particolari espedienti permettono che script con funzionalità non specifiche possano essere eseguiti sia in modalità interattiva che non.

Se uno script ha bisogno di verificare se è in esecuzione in una shell interattiva, basta semplicemente controllare se la variabile del *prompt*, $PS1, è impostata. (Se l'utente dev'essere pronto ad inserire un input allora lo script deve visualizzare un prompt.)

```
 1 if [ -z $PS1 ] # nessun prompt?
 2 then
 3   # non interattiva
 4   ...
 5 else
 6   # interattiva
 7   ...
 8 fi
```

Alternativamente, lo script può verificare la presenza dell'opzione "i" in $-.

```
 1 case $- in
```

```
 2 *i*)    # shell interattiva
 3 ;;
 4 *)      # shell non interattiva
 5 ;;
 6 # (Cortesia di "UNIX F.A.Q.," 1993)
```

È possibile forzare l'esecuzione degli script in modalità interattiva con l'opzione -i o con l'intestazione **#!/bin/bash -i**. Si faccia però attenzione che questo potrebbe causare un comportamento irregolare dello script o visualizzare messaggi d'errore anche quando non ve ne sono.

34.2. Shell wrapper

Un "wrapper" è uno script di shell che incorpora una utility o un comando di sistema. Questo evita di dover digitare una serie di parametri che andrebbero passati manualmente a quel comando. "Avvolgere" uno script attorno ad una complessa riga di comando ne semplifica l'invocazione. Questo è particolarmente utile con sed e awk.

Uno script **sed** o **awk**, di norma, dovrebbe essere invocato da riga di comando con **sed -e *'comandi'*** o **awk *'comandi'***. Inserire un tale script in uno script Bash permette di richiamarlo in modo più semplice, rendendolo anche "riutilizzabile". Così è anche possibile combinare le funzionalità di **sed** e **awk**, per esempio collegando con una pipe l'output di una serie di comandi sed a awk. Se salvato come file eseguibile può essere ripetutamente invocato, nella sua forma originale o modificata, senza l'inconveniente di doverlo ridigitare completamente da riga di comando.

Esempio 34-1. Shell wrapper

```
 1 #!/bin/bash
 2
 3 # Questo è un semplice script che rimuove le righe vuote da un file.
 4 # Nessuna verifica d'argomento.
 5 #
 6 # Sarebbe meglio aggiungere qualcosa come:
 7 # if [ -z "$1" ]
 8 # then
 9 #  echo "Utilizzo: `basename $0` nome-file"
10 #  exit 65
11 # fi
12
13
14 # È uguale a
15 #    sed -e '/^$/d' nomefile
16 # invocato da riga di comando.
17
18 sed -e /^$/d "$1"
19 #  '-e' significa che segue un comando di "editing" (in questo caso
opzionale).
20 #  '^' indica l'inizio della riga, '$' la fine.
21 #  Verifica le righe che non contengono nulla tra il loro inizio e la fine,
22 #+ vale a dire, le righe vuote.
23 #  'd' è il comando di cancellazione.
24
25 #  L'uso del quoting per l'argomento consente di
26 #+ passare nomi di file contenenti spazi e caratteri speciali.
27
```

```
28 exit 0
```

Esempio 34-2. Uno shell wrapper leggermente più complesso

```
 1 #!/bin/bash
 2
 3 #  "subst", uno script per sostituire un nome
 4 #+ con un altro all'interno di un file,
 5 #+ es., "subst Smith Jones letter.txt".
 6
 7 ARG=3          # Lo script richiede tre argomenti.
 8 E_ERR_ARG=65   # Numero errato di argomenti passati allo script.
 9
10 if [ $# -ne "$ARG" ]
11 # Verifica il numero degli argomenti (è sempre una buona idea).
12 then
13   echo "Utilizzo: `basename $0` vecchio-nome nuovo-nome nomefile"
14   exit $E_ERR_ARG
15 fi
16
17 vecchio_nome=$1
18 nuovo_nome=$2
19
20 if [ -f "$3" ]
21 then
22   nome_file=$3
23 else
24   echo "Il file \"$3\" non esiste."
25   exit $E_ERR_ARG
26 fi
27
28 #  Ecco dove viene svolto il lavoro principale.
29
30 # ------------------------------------------------
31 sed -e "s/$vecchio_nome/$nuovo_nome/g" $nome_file
32 # ------------------------------------------------
33
34 #  's' è, naturalmente, il comando sed di sostituzione,
35 #+ e /modello/ invoca la ricerca di corrispondenza.
36 #  L'opzione "g", o globale, provoca la sostituzione di *tutte*
37 #+ le occorrenze di $vecchio_nome in ogni riga, non solamente nella prima.
38 #  Leggete i testi riguardanti 'sed' per una spiegazione più approfondita.
39
40 exit 0    # Lo script invocato con successo restituisce 0.
```

Esempio 34-3. Uno shell wrapper per uno script awk

```
 1 #!/bin/bash
 2
 3 # Aggiunge la colonna specificata (di numeri) nel file indicato.
 4
 5 ARG=2
 6 E_ERR_ARG=65
 7
 8 if [ $# -ne "$ARG" ] #  Verifica il corretto nr. di argomenti da riga
 9                       #+ di comando.
10 then
11    echo "Utilizzo: `basename $0` nomefile numero-colonna"
12    exit $E_ERR_ARG
13 fi
```

```
14
15 nomefile=$1
16 numero_colonna=$2
17
18 # Il passaggio di variabili di shell ad awk, che è parte dello script
19 #+ stesso, è un po' delicato.
20 # Vedete la documentazione awk per maggiori dettagli.
21
22 # Uno script awk che occupa più righe viene invocato con   awk ' ..... '
23
24
25 # Inizio dello script awk.
26 # ----------------------------
27 awk '
28
29 { totale += $'"${numero_colonna}"'
30 }
31 END {
32       print totale
33 }
34
35 ' "$nomefile"
36 # ----------------------------
37 # Fine dello script awk.
38
39
40 #   Potrebbe non essere sicuro passare variabili di shell a uno script awk
41 #   incorporato, così Stephane Chazelas propone la seguente alternativa:
42 #   -------------------------------------
43 #   awk -v numero_colonna="$numero_colonna" '
44 #   { totale += $numero_colonna
45 #   }
46 #   END {
47 #       print totale
48 #   }' "$nomefile"
49 #   -------------------------------------
50
51
52 exit 0
```

Per quegli script che necessitano di un unico strumento tuttofare, un coltellino svizzero informatico, esiste Perl. Perl combina le capacità di **sed** e **awk**, e, per di più, un'ampia parte di quelle del **C**. È modulare e supporta qualsiasi cosa, dalla programmazione orientata agli oggetti fino alla preparazione del caffè. Brevi script in Perl si prestano bene ad essere inseriti in script di shell e si può anche dichiarare, con qualche ragione, che Perl possa sostituire completamente lo scripting di shell stesso (sebbene l'autore di questo documento rimanga scettico).

Esempio 34-4. Perl inserito in uno script Bash

```
 1 #!/bin/bash
 2
 3 # I comandi shell possono precedere lo script Perl.
 4 echo "Questa riga precede lo script Perl inserito in \"$0\"."
 5 echo "============================================================="
 6
 7 perl -e 'print "Questo è lo script Perl che è stato inserito.\n";'
 8 # Come sed, anche Perl usa l'opzione "-e".
 9
10 echo "============================================================="
11 echo "Comunque, lo script può contenere anche comandi di shell e di
```

```
sistema."
 12
 13 exit 0
```

È anche possibile combinare, in un unico file, uno script Bash e uno script Perl. Dipenderà dal modo in cui lo script verrà invocato quale delle due parti sarà eseguita.

Esempio 34-5. Script Bash e Perl combinati

```
  1 #!/bin/bash
  2 # bashandperl.sh
  3
  4 echo "Saluti dalla parte Bash dello script."
  5 # Qui possono seguire altri comandi Bash.
  6
  7 exit 0
  8 # Fine della parte Bash dello script.
  9
 10 # ===========================================================
 11
 12 #!/usr/bin/perl
 13 # Questa parte dello script deve essere invocata con l'opzione -x.
 14
 15 print "Saluti dalla parte Perl dello script.\n";
 16 # Qui possono seguire altri comandi Perl.
 17
 18 # Fine della parte Perl dello script.
```

```
bash$ bash bashandperl.sh
Saluti dalla parte Bash dello script.


bash$ perl -x bashandperl.sh
Saluti dalla parte Perl dello script.
```

34.3. Verifiche e confronti: alternative

Per le verifiche è più appropriato il costrutto [[ ]] che non con **[ ]**. Lo stesso vale per il costrutto (( )) per quanto concerne i confronti aritmetici.

```
  1 a=8
  2
  3 # Tutti i confronti seguenti si equivalgono.
  4 test "$a" -lt 16 && echo "sì, $a < 16"         #  "lista and"
  5 /bin/test "$a" -lt 16 && echo "sì, $a < 16"
  6 [ "$a" -lt 16 ] && echo "sì, $a < 16"
  7 [[ $a -lt 16 ]] && echo "sì, $a < 16"          #  Non è necessario il
quoting
  8                                                #+ delle variabili presenti
in [[ ]] e (( )).
  9 (( a < 16 )) && echo "sì, $a < 16"
 10
 11 città="New York"
 12 # Anche qui, tutti i confronti seguenti si equivalgono.
 13 test "$città" \< Parigi && echo "Sì, Parigi è più grande di $città"
 14 # Più grande in ordine ASCII.
```

```
 15 /bin/test "$città" \< Parigi && echo "Sì, Parigi è più grande di $città"
 16 [ "$città" \< Parigi ] && echo "Sì, Parigi è più grande di $città"
 17 [[ $città < Parigi ]] && echo "Sì, Parigi è più grande di $città"
 18 # $città senza quoting.
 19
 20 # Grazie, S.C.
```

34.4. Ricorsività

Può uno script richiamare sé stesso ricorsivamente? Certo.

Esempio 34-6. Un (inutile) script che richiama sé stesso ricorsivamente

```
  1 #!/bin/bash
  2 # recurse.sh
  3
  4 #  Può uno script richiamare sé stesso ricorsivamente?
  5 #  Sì, ma può essere di qualche uso pratico?
  6 #  (Vedi il successivo.)
  7
  8 INTERVALLO=10
  9 VALMAX=9
 10
 11 i=$RANDOM
 12 let "i %= $INTERVALLO"  #  Genera un numero casuale compreso
 13                         #+ tra 0 e $INTERVALLO - 1.
 14
 15 if [ "$i" -lt "$VALMAX" ]
 16 then
 17   echo "i = $i"
 18   ./$0                  #  Lo script genera ricorsivamente una nuova istanza
 19                         #+ di sé stesso.
 20 fi                      #  Ogni script figlio fa esattamente la stessa
 21                         #+ cosa, finché un $i non sia uguale a $VALMAX.
 22
 23 #  L'uso di un ciclo "while", invece della verifica "if/then", provoca
problemi.
 24 #  Spiegate perché.
 25
 26 exit 0
 27
 28 # Nota:
 29 # ----
 30 # Lo script, per funzionare correttamente, deve avere il permesso di
esecuzione.
 31 # Questo anche nel caso in cui venga invocato con il comando "sh".
 32 # Spiegate perchè.
```

Esempio 34-7. Un (utile) script che richiama sé stesso ricorsivamente

```
  1 #!/bin/bash
  2 # pb.sh: phone book
  3
  4 # Scritto da Rick Boivie e usato con il consenso dell'autore.
  5 # Modifiche effettuate dall'autore del documento.
  6
  7 MINARG=1      #  Lo script ha bisogno di almeno un argomento.
```

```
 8 FILEDATI=./phonebook
 9               #  Deve esistere un file dati di nome "phonebook"
10               #+ nella directory di lavoro corrente.
11 NOMEPROG=$0
12 E_NON_ARG=70  #  Errore di nessun argomento.
13
14 if [ $# -lt $MINARG ]; then
15       echo "Utilizzo: "$NOMEPROG" filedati"
16       exit $E_NON_ARG
17 fi
18
19
20 if [ $# -eq $MINARG ]; then
21       grep $1 "$FILEDATI"
22       # 'grep' visualizza un messaggio d'errore se $FILEDATI non esiste.
23 else
24       ( shift; "$NOMEPROG" $* ) | grep $1
25       # Lo script richiama sé stesso ricorsivamente.
26 fi
27
28 exit 0        #  Lo script termina qui.
29               #  Quindi, è corretto mettere
30               #+ dati e commenti senza il # oltre questo punto.
31
32 # ------------------------------------------------------------------------
33 # Un estratto del file dati "phonebook":
34
35 John Doe        1555 Main St., Baltimore, MD 21228           (410) 222-3333
36 Mary Moe        9899 Jones Blvd., Warren, NH 03787           (603) 898-3232
37 Richard Roe     856 E. 7th St., New York, NY 10009           (212) 333-4567
38 Sam Roe         956 E. 8th St., New York, NY 10009           (212) 444-5678
39 Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338   (415) 501-1631
40 # ------------------------------------------------------------------------
41
42 $bash pb.sh Roe
43 Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
44 Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678
45
46 $bash pb.sh Roe Sam
47 Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678
48
49 #  Quando vengono passati più argomenti allo script,
50 #+ viene visualizzata *solo* la/e riga/he contenente tutti gli argomenti.
```

Esempio 34-8. Un altro (utile) script che richiama sé stesso ricorsivamente

```
 1 #!/bin/bash
 2 # usrmnt.sh, scritto da Anthony Richardson
 3 # Utilizzato con il permesso dell'autore.
 4
 5 # utilizzo:    usrmnt.sh
 6 # descrizione: monta un dispositivo, l'utente cho lo invoca deve essere
elencato
 7 #              nel gruppo MNTUSERS nel file /etc/sudoers.
 8
 9 # ---------------------------------------------------------------------
10 #  Si tratta dello script usermount che riesegue se stesso usando sudo.
11 #  Un utente con i permessi appropriati deve digitare semplicemente
12
13 #   usermount /dev/fd0 /mnt/floppy
14
```

```
15 # invece di
16
17 #   sudo usermount /dev/fd0 /mnt/floppy
18
19 #  Utilizzo questa tecnica per tutti gli
20 #+ script sudo perché la trovo conveniente.
21 # ------------------------------------------------------------------
22
23 #  Se la variabile SUDO_COMMAND non è impostata, significa che non lo si
24 #+ sta eseguendo attraverso sudo, che quindi va richiamato. Vengono passati
25 #+ i veri id utente e di gruppo . . .
26
27 if [ -z "$SUDO_COMMAND" ]
28 then
29    mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
30    exit 0
31 fi
32
33 # Verrà eseguita questa riga solo se lo si sta eseguendo con sudo.
34 /bin/mount $* -o uid=$mntusr,gid=$grpusr
35
36 exit 0
37
38 # Note aggiuntive (dell'autore dello script):
39 # -----------------------------------------------
40
41 # 1) Linux consente l'uso dell'opzione "users" nel file /etc/fstab,
42 #    quindi qualsiasi utente può montare un certo dispositivo.
43 #    Ma, su un server, è preferibile consentire l'accesso ai dispositivi
44 #    solo a pochi individui.
45 #    Trovo che usare sudo dia un maggior controllo.
46
47 # 2) Trovo anche che, per ottenere questo risultato, sudo sia più
48 #    conveniente che utilizzare i gruppi.
49
50 # 3) Questo metodo fornisce, a tutti coloro dotati dei corretti permessi,
51 #    l'accesso root al comando mount, quindi fate attenzione a chi lo
52 #    concedete.
53 #    È possibile ottenere un controllo ancora più preciso
54 #    utilizzando questa tecnica in differenti script ciascono inerente a
55 #    mntfloppy, mntcdrom e mntsamba.
```


Troppi livelli di ricorsività possono esaurire lo spazio di stack dello script, provocando un segmentation fault

34.5. "Colorare" con gli script

Le sequenze di escape ANSI [1] impostano gli attributi dello schermo, come il testo in grassetto e i colori del primo piano e dello sfondo. I file batch DOS usano comunemente i codici di escape ANSI per *colorare* i loro output, e altrettanto possono fare gli script Bash.

Esempio 34-9. Una rubrica di indirizzi "a colori"

```
1 #!/bin/bash
2 # ex30a.sh: Versione di ex30.sh "a colori".
3 #           Un database di indirizzi non molto elegante
4
5
```

```
 6 clear                                   # Pulisce lo schermo.
 7
 8 echo -n "          "
 9 echo -e '\E[37;44m'"\033[1mElenco Contatti\033[0m"
10                                         # Bianco su sfondo blu
11 echo; echo
12 echo -e "\033[1mScegliete una delle persone seguenti:\033[0m"
13                                         # Grassetto
14 tput sgr0
15 echo "(Inserite solo la prima lettera del nome.)"
16 echo
17 echo -en '\E[47;34m'"\033[1mE\033[0m"   # Blu
18 tput sgr0                               # Ripristina i colori "normali."
19 echo "vans, Roland"                     # "[E]vans, Roland"
20 echo -en '\E[47;35m'"\033[1mJ\033[0m"   # Magenta
21 tput sgr0
22 echo "ones, Mildred"
23 echo -en '\E[47;32m'"\033[1mS\033[0m"   # Verde
24 tput sgr0
25 echo "mith, Julie"
26 echo -en '\E[47;31m'"\033[1mZ\033[0m"   # Rosso
27 tput sgr0
28 echo "ane, Morris"
29 echo
30
31 read persona
32
33 case "$persona" in
34 # Notate l'uso del "quoting" per la variabile.
35
36   "E" | "e" )
37   # Accetta sia una lettera maiuscola che una minuscola.
38   echo
39   echo "Roland Evans"
40   echo "4321 Floppy Dr."
41   echo "Hardscrabble, CO 80753"
42   echo "(303) 734-9874"
43   echo "(303) 734-9892 fax"
44   echo "revans@zzy.net"
45   echo "Socio d'affari & vecchio amico"
46   ;;
47
48   "J" | "j" )
49   echo
50   echo "Mildred Jones"
51   echo "249 E. 7th St., Apt. 19"
52   echo "New York, NY 10009"
53   echo "(212) 533-2814"
54   echo "(212) 533-9972 fax"
55   echo "milliej@loisaida.com"
56   echo "Fidanzata"
57   echo "Compleanno: Feb. 11"
58   ;;
59
60 # Aggiungete in seguito le informazioni per Smith & Zane.
61
62           * )
63    # Opzione preefinita.
64    # Anche un input vuoto (è stato premuto il tasto INVIO) viene verificato
qui.
65    echo
66    echo "Non ancora inserito nel database."
```

```
67   ;;
68
69 esac
70
71 tput sgr0                                # Ripristina i colori "normali."
72
73 echo
74
75 exit 0
```

Esempio 34-10. Disegnare un rettangolo

```
 1 #!/bin/bash
 2 # Draw-box.sh: Disegnare un rettangolo con caratteri ASCII.
 3
 4 # Script di Stefano Palmeri, con modifiche secondarie dell'autore del libro.
 5 # Usato in "Guida ABS" con il consenso dell'autore dello script.
 6
 7
 8 ##########################################################################
 9 ###  spiegazione della funzione disegna_rettangolo  ###
10
11 #  La funzione "disegna_rettangolo" permette all'utente
12 #+ di disegnare un rettangolo in un terminale.
13 #
14 #  Utilizzo: disegna_rettangolo RIGA COLONNA ALTEZZA BASE [COLORE]
15 #  RIGA e COLONNA rappresentano la posizione
16 #+ dell'angolo superiore sinistro del rettangolo da disegnare.
17 #  RIGA e COLONNA devono essre maggiori di 0
18 #+ e minori della dimensione del terminale corrente.
19 #  ALTEZZA è il numero di righe del rettangolo, e deve essere > 0.
20 #  ALTEZZA + RIGA deve essere <= dell'altezza del terminale corrente.
21 #  BASE è il numero di colonne del rettangolo e deve essere > 0.
22 #  BASE + COLONNA deve essere <= dell'ampiezza del terminale corrente.
23 #
24 # Es.: se la dimensione del terminale fosse di 20x80,
25 #  disegna_rettangolo 2 3 10 45 andrebbe bene
26 #  disegna_rettangolo 2 3 19 45 valore di ALTEZZA errato (19+2 > 20)
27 #  disegna_rettangolo 2 3 18 78 valore di BASE errato (78+3 > 80)
28 #
29 #  COLORE è il colore dei lati del rettangolo.
30 #  È il 5° argomento ed è opzionale.
31 #  0=nero 1=rosso 2=verde 3=marrone 4=blu 5=porpora 6=cyan 7=bianco.
32 #  Se alla funzione viene passato un numero di argomenti errato,
33 #+ lo script termina con il codice d'errore 65
34 #+ e nessun messaggio verrà visualizzato allo stderr.
35 #
36 #  Pulite lo schermo prima di iniziare a disegnare un rettangolo.
37 #  Il comando clear non è presente nella funzione.
38 #  Questo per consentire all'utente di disegnare più rettangoli,
39 #+ anche sovrapponendoli.
40
41 ###  fine della spiegazione della funzione disegna_rettangolo  ###
42 ##########################################################################
43
44 disegna_rettangolo(){
45
46 #=============#
47 ORIZ="-"
48 VERT="|"
49 ANGOLO="+"
```

```
 50
 51 ARGMIN=4
 52 E_ERRARG=65
 53 #=============#
 54
 55
 56 if [ $# -lt "$ARGMIN" ]; then                 #  Se gli argomenti sono meno
 57     exit $E_ERRARG                           #+ di 4, esce.
 58 fi
 59
 60 # Controlla che gli argomenti siano solo dei numeri.
 61 # Probabilmente potrebbe essere fatto meglio (esercizio per il lettore?).
 62 if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
 63    exit $E_ERRARG
 64 fi
 65
 66 ALTEZZA_RET=`expr $3 - 1`  #  -1 correzione necessaria perchè il carattere
per
 67 AMPIEZZA_RET=`expr $4 - 1` #+ gli angoli "+" fa parte sia dell'altezza che
della
 68                           #+ larghezza.
 69 RIGHE_T=`tput lines`       #  Si determina la dimensione del terminale
corrente
 70 COL_T=`tput cols`          #+ in numero di righe e colonne.
 71
 72 if [ $1 -lt 1 ] || [ $1 -gt $RIGHE_T ]; then   #  Inizio delle verifiche di
 73    exit $E_ERRARG                              #+ congruità degli argomenti.
 74 fi
 75 if [ $2 -lt 1 ] || [ $2 -gt $COL_T ]; then
 76    exit $E_ERRARG
 77 fi
 78 if [ `expr $1 + $ALTEZZA_RET + 1` -gt $RIGHE_T ]; then
 79    exit $E_BADARGS
 80 fi
 81 if [ `expr $2 + $AMPIEZZA_RET + 1` -gt $COL_T ]; then
 82    exit $E_ERRARG
 83 fi
 84 if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
 85    exit $E_ERRARG
 86 fi                                  # Fine delle verifiche degli argomenti.
 87
 88 vis_car(){                          # Funzione all'interno di una funzione.
 89    echo -e "\E[${1};${2}H"$3
 90 }
 91
 92 echo -ne "\E[3${5}m"               #  Imposta il colore del rettangolo,
 93                                    #+ se specificato.
 94
 95 # inizia il disegno del rettangolo
 96
 97 conto=1                                          #  Disegna le righe
verticali
 98 for (( r=$1; conto<=$ALTEZZA_RET; r++)); do     #+ con la funzione vis_car.
 99   vis_car $r $2 $VERT
100   let conto=conto+1
101 done
102
103 conto=1
104 c=`expr $2 + $AMPIEZZA_RET`
105 for (( r=$1; conto<=$ALTEZZA_RET; r++)); do
106   vis_car $r $c $VERT
107   let conto=conto+1
```

```
108 done
109
110 conto=1                                        #  Disegna le righe
orizzontali
111 for (( c=$2; conto<=$AMPIEZZA_RET; c++)); do   #+ con la funzione vis_car.
112   vis_car $1 $c $ORIZ
113   let conto=conto+1
114 done
115
116 conto=1
117 r=`expr $1 + $ALTEZZA_RET`
118 for (( c=$2; conto<=$AMPIEZZA_RET; c++)); do
119   vis_car $r $c $ORIZ
120   let conto=conto+1
121 done
122
123 vis_car $1 $2 $ANGOLO                          # Inserisce gli angoli.
124 vis_car $1 `expr $2 + $AMPIEZZA_RET` +
125 vis_car `expr $1 + $ALTEZZA_RET` $2 +
126 vis_car `expr $1 + $ALTEZZA_RET` `expr $2 + $AMPIEZZA_RET` +
127
128 echo -ne "\E[0m"             #  Ripristina i colori precedenti.
129
130 RIGHE_P=`expr $RIGHE_T - 1`  #  Posiziona il prompt in fondo al terminale.
131
132 echo -e "\E[${RIGHE_P};1H"
133 }
134
135
136 # Ora proviamo a disegnare il rettangolo.
137 clear                                          # Pulisce il terminale.
138 R=2      # Righe
139 C=3      # Colonne
140 A=10     # Altezza
141 L=45     # Larghezza
142 col=1    # Colore (rosso)
143 disegna_rettangolo $R $C $A $L $col           # Disegna il rettangolo.
144
145 exit 0
146
147 # Esercizio:
148 # ---------
149 # Aggiungete l'opzione per inserire del testo nel rettangolo.
```

La più semplice e, forse, più utile sequenza di escape ANSI è quella per l'impostazione del testo in grassetto, **\033[1m ... \033[0m**. \033 rappresenta un *escape*, "[1" abilita l'attributo del grassetto, mentre "[0" lo disabilita. "m" indica la fine di ogni termine della sequenza di escape.

```
bash$ echo -e "\033[1mQuesto testo è in grassetto.\033[0m"
```

Una sequenza simile abilita l'attributo di sottolineatura (su terminali *rxvt* e *aterm*).

```
bash$ echo -e "\033[4mQuesto testo è sottolineato.\033[0m"
```

L'opzione `-e` di **echo** abilita le sequenze di escape.

Altre sequenze modificano il colore del testo e/o dello sfondo.

```
bash$ echo -e '\E[34;47mQuesto viene visualizzato in blu.'; tput sgr0

bash$ echo -e '\E[33;44m'"Testo giallo su sfondo blu."; tput sgr0

bash$ echo -e '\E[1;33;44m'"Testo giallo in GRASSETTO" su sfondo blu.; tput sgr0
```

 Di solito è consigliabile impostare l'attributo di *grassetto* per il testo colorato in primo piano.

tput sgr0 ripristina il terminale alle normali impostazioni. Se viene omesso, tutti i successivi output, su quel particolare terminale, rimarranno blu.

 Poiché **tput sgr0**, in certe circostanze, fallisce nel ripristinare le precedenti impostazioni, **echo -ne \E[0m** potrebbe rivelarsi una scelta migliore.

Si utilizzi il seguente schema per scrivere del testo colorato su uno sfondo altrettanto colorato.

```
echo -e '\E[COLORE1;COLORE2mQui va inserito il testo.'
```

"\E[" da inizio alla sequenza di escape. I numeri corrispondenti a "COLORE1" e "COLORE2", separati dal punto e virgola, specificano i colori di primo piano e dello sfondo, secondo i valori indicati nella tabella riportata più sotto. (L'ordine dei numeri non è importante perché quelli per il primo piano cadono in un intervallo che non si sovrappone a quello dei numeri dello sfondo.) "m" termina la sequenza di escape ed il testo deve incominciare immediatamente dopo.

Si noti che tutta la sequenza di escape che viene dopo **echo -e** va racchiusa tra apici singoli.

I numeri della seguente tabella valgono per un terminale *rxvt*. I risultati potrebbero variare su altri emulatori di terminale.

Tabella 34-1. Numeri che rappresentano i colori nelle sequenze di escape

Colore	Primo piano	Sfondo
nero	30	40
rosso	31	41
verde	32	42
giallo	33	43
blu	34	44
magenta	35	45
cyan	36	46
bianco	37	47

Esempio 34-11. Visualizzare testo colorato

```
 1 #!/bin/bash
 2 # color-echo.sh: Visualizza messaggi colorati.
 3
 4 # Modificate lo script secondo le vostre necessità.
```

```
 5 # Più facile che codificare i colori.
 6
 7 nero='\E[30;47m'
 8 rosso='\E[31;47m'
 9 verde='\E[32;47m'
10 giallo='\E[33;47m'
11 blu='\E[34;47m'
12 magenta='\E[35;47m'
13 cyan='\E[36;47m'
14 bianco='\E[37;47m'
15
16
17 alias Reset="tput sgr0"      #  Ripristina gli attributi di testo normali
18                              #+ senza pulire lo schermo.
19
20
21 cecho ()                     # Colora-echo.
22                              # Argomento $1 = messaggio
23                              # Argomento $2 = colore
24 {
25 local msg_default="Non è stato passato nessun messaggio."
26                              #  Veramente, non ci sarebbe bisogno di una
27                              #+ variabile locale.
28
29 messaggio=${1:-$msg_default} #  Imposta al messaggio predefinito se non ne
30                              #+ viene fornito alcuno.
31 colore=${2:-$nero}           #  Il colore preimpostato è il nero, se
32                              #+ non ne viene specificato un altro.
33
34   echo -e "$colore"
35   echo "$messaggio"
36   Reset                      # Ripristina i valori normali.
37
38   return
39 }
40
41
42 # Ora lo mettiamo alla prova.
43 # -------------------------------------------------
44 cecho "Mi sento triste..." $blu
45 cecho "Il magenta assomiglia molto al porpora." $magenta
46 cecho "Sono verde dall'invidia." $verde
47 cecho "Vedi rosso?" $rosso
48 cecho "Cyan, più familiarmente noto come acqua." $cyan
49 cecho "Non è stato passato nessun colore (nero di default)."
50        # Omesso l'argomento $colore.
51 cecho "Il colore passato è \"nullo\" (nero di default)." ""
52        # Argomento $colore nullo.
53 cecho
54        # Omessi gli argomenti $messaggio e $colore.
55 cecho "" ""
56        # Argomenti $messaggio e $colore nulli.
57 # -------------------------------------------------
58
59 echo
60
61 exit 0
62
63 # Esercizi:
64 # ---------
65 # 1) Aggiungete l'attributo "grassetto" alla funzione 'cecho ()'.
66 # 2) Aggiungete delle opzioni per colorare gli sfondi.
```

Esiste, comunque, un grosso problema. *Le sequenze di escape ANSI non sono assolutamente portabili*. Ciò che funziona bene su certi emulatori di terminale (o sulla console) potrebbe funzionare in modo diverso (o per niente) su altri. Uno script "a colori" che appare sbalorditivo sulla macchina del suo autore, potrebbe produrre un output illeggibile su quella di qualcun altro. Questo fatto compromette grandemente l'utilità di "colorazione" degli script, relegando questa tecnica allo stato di semplice espediente o addirittura di "bazzecola".

L'utility **color** di Moshe Jacobson (http://runslinux.net/projects.html#color) semplifica considerevolmente l'uso delle sequenze di escape ANSI. Essa sostituisce i goffi costrutti appena trattati con una sintassi chiara e logica.

Note

[1] Naturalmente, ANSI è l'acronimo di American National Standards Institute.

34.6. Ottimizzazioni

La maggior parte degli script di shell rappresentano delle soluzioni rapide e sommarie per problemi non troppo complessi. Come tali, la loro ottimizzazione, per una esecuzione veloce, non è una questione importante. Si consideri il caso, comunque, di uno script che esegue un compito rilevante, lo fa bene, ma troppo lentamente. Riscriverlo in un linguaggio compilato potrebbe non essere un'opzione accettabile. La soluzione più semplice consiste nel riscrivere le parti dello script che ne rallentano l'esecuzione. È possibile applicare i principi di ottimizzazione del codice anche ad un modesto script di shell?

Si controllino i cicli dello script. Il tempo impiegato in operazioni ripetitive si somma rapidamente. Per quanto possibile, si tolgano dai cicli le operazioni maggiormente intensive in termini di tempo.

È preferibile usare i comandi builtin invece dei comandi di sistema. I builtin vengono eseguiti più velocemente e, di solito, non lanciano, quando vengono invocati, delle subshell.

Si evitino i comandi inutili, in modo particolare nelle pipe.

```
1 cat "$file" | grep "$parola"
2
3 grep "$parola" "$file"
4
5 #  Le due linee di comando hanno un effetto identico, ma la seconda viene
6 #+ eseguita più velocemente perché lancia un sottoprocesso in meno.
```

Sembra che, negli script, ci sia la tendenza ad abusare del comando cat.

Si usino time e times per calcolare il tempo impiegato nell'esecuzione dei comandi. Si prenda in considerazione la possibilità di riscrivere sezioni di codice critiche, in termini di tempo, in C, se non addirittura in assembler.

Si cerchi di minimizzare l'I/O di file. Bash non è particolarmente efficiente nella gestione dei file. Si consideri, quindi, l'impiego di strumenti più appropriati allo scopo, come awk o Perl.

Si scrivano gli script in forma coerente e strutturata, così che possano essere riorganizzati e ridotti in caso di necessità. Alcune delle tecniche di ottimizzazione applicabili ai linguaggi di alto livello possono funzionare anche per gli script, ma altre, come lo svolgimento del ciclo, sono per lo più irrilevanti. Soprattutto, si usi il buon senso.

Per un'eccellente dimostrazione di come l'ottimizzazione possa ridurre drasticamente il tempo di esecuzione di uno script, vedi Esempio 12-38

34.7. Argomenti vari

- Per mantenere una registrazione di quali script sono stati eseguiti durante una particolare sessione, o un determinato numero di sessioni, si aggiungano le righe seguenti a tutti gli script di cui si vuole tener traccia. In questo modo verrà continuamente aggiornato il file di registrazione con i nomi degli script e con l'ora in cui sono stati posti in esecuzione.

```
 1 # Accodatte (>>) le righe seguenti alla fine di ogni script di cui
 2 #+ volete tener traccia.
 3
 4 whoami>> $SAVE_FILE    # Utente che ha invocato lo script.
 5 echo $0>> $SAVE_FILE   # Nome dello script.
 6 date>> $SAVE_FILE      # Data e ora.
 7 echo>> $SAVE_FILE      # Riga bianca di separazione.
 8
 9 #  Naturalmente, SAVE_FILE deve essere definito ed esportato come
10 #+ variabile d'ambiente in ~/.bashrc
11 #+ (qualcosa come ~/.script-eseguiti)
```

- L'operatore >> accoda delle righe in un file. Come si può fare, invece, se si desidera *anteporre* una riga in un file esistente, cioè, inserirla all'inizio?

```
 1 file=dati.txt
 2 titolo="***Questa è la riga del titolo del file di testo dati***"
 3
 4 echo $titolo | cat - $file >$file.nuovo
 5 # "cat -" concatena lo stdout a $file.
 6 #  Il risultato finale è
 7 #+ la creazione di un nuovo file con $titolo aggiunto all'*inizio*.
```

- È una variante semplificata dello script di Esempio 17-13 già visto in precedenza. Naturalmente, anche sed è in grado di svolgere questo compito.
- Uno script di shell può agire come un comando inserito all'interno di un altro script di shell, in uno script *Tcl* o *wish*, o anche in un Makefile. Può essere invocato come un comando esterno di shell in un programma C, per mezzo della funzione `system()`, es. `system("nome_script");`.
- Si raggruppino in uno o più file le funzioni e le definizioni preferite e più utili. Al bisogno, si può "includere" uno o più di questi "file libreria" negli script per mezzo sia del punto (.) che del comando source.

```
 1 # LIBRERIA PER SCRIPT
 2 # ------ ------- ------
 3
 4 # Nota:
 5 # Non è presente "#!"
```

```
  6 # Né "codice eseguibile".
  7
  8
  9 # Definizioni di variabili utili
 10
 11 UID_ROOT=0             # Root ha $UID 0.
 12 E_NONROOT=101          # Errore di utente non root.
 13 MAXVALRES=255          # Valore di ritorno massimo (positivo) di una
funzione.
 14 SUCCESSO=0
 15 INSUCCESSO=-1
 16
 17
 18 # Funzioni
 19
 20 Utilizzo ()            # Messaggio "Utilizzo:".
 21 {
 22   if [ -z "$1" ]       # Nessun argomento passato.
 23   then
 24      msg=nomefile
 25   else
 26      msg=$@
 27   fi
 28
 29   echo "Utilizzo: `basename $0` "$msg""
 30 }
 31
 32
 33 Controlla_root ()      # Controlla se è root ad eseguire lo script.
 34 {                      # Dall'esempio "ex39.sh".
 35   if [ "$UID" -ne "$UID_ROOT" ]
 36   then
 37      echo "Devi essere root per eseguire questo script."
 38      exit $E_NONROOT
 39   fi
 40 }
 41
 42
 43 CreaNomeFileTemp ()    # Crea un file temporaneo con nome "unico".
 44 {                      # Dall'esempio "ex51.sh".
 45   prefisso=temp
 46   suffisso=`eval date +%s`
 47   Nomefiletemp=$prefisso.$suffisso
 48 }
 49
 50
 51 isalpha2 ()            # Verifica se l'*intera stringa* è formata da
 52                        #+ caratteri alfabetici.
 53 {                      # Dall'esempio "isalpha.sh".
 54   [ $# -eq 1 ] || return $INSUCCESSO
 55
 56   case $1 in
 57   *[!a-zA-Z]*|"") return $INSUCCESSO;;
 58   *) return $SUCCESSO;;
 59   esac                 # Grazie, S.C.
 60 }
 61
 62
 63 abs ()                           # Valore assoluto.
 64 {                                # Attenzione: Valore di ritorno
massimo = 255.
 65   E_ERR_ARG=-999999
```

```
 66
 67   if [ -z "$1" ]                # È necessario passare un
argomento.
 68   then
 69      return $E_ERR_ARG          # Ovviamente viene restituito il
 70                                 #+ codice d'errore.
 71   fi
 72
 73   if [ "$1" -ge 0 ]             # Se non negativo,
 74   then                          #
 75      valass=$1                  # viene preso così com'è.
 76   else                          # Altrimenti,
 77      let "valass = (( 0 - $1 ))" # cambia il segno.
 78   fi
 79
 80   return $valass
 81 }
 82
 83
 84 in_minuscolo ()        #  Trasforma la/e stringa/he passata/e come
argomento/i
 85 {                      #+ in caratteri minuscoli.
 86
 87   if [ -z "$1" ]       #  Se non viene passato alcun argomento,
 88   then                 #+ invia un messaggio d'errore
 89      echo "(null)"     #+ (messaggio d'errore di puntatore vuoto in
stile C)
 90      return           #+ e uscita dalla funzione.
 91   fi
 92
 93   echo "$@" | tr A-Z a-z
 94   # Modifica di tutti gli argomenti passati ($@).
 95
 96   return
 97
 98 #  Usate la sostituzione di comando per impostare una variabile
all'output
 99 #+ della funzione.
100 #  Per esempio:
101 #    vecchiavar="unA seRiE di LEtTerE mAiUscoLe e MInusColE
MisCHiaTe"
102 #    nuovavar=`in_minuscolo "$vecchiavar"`
103 #    echo "$nuovavar"  # una serie di lettere maiuscole e minuscole
mischiate
104 #
105 #  Esercizio: Riscrivete la funzione per modificare le lettere
minuscole del/gli
106 #+ argomento/i passato/i in lettere maiuscole ... in_maiuscolo()
[facile].
107
108 }
```

- Si utilizzino intestazioni di commento particolareggiate per aumentare la chiarezza e la leggibilità degli script.

```
  1 ## Attenzione.
  2 rm -rf *.zzy   ##  Le opzioni "-rf" di "rm" sono molto pericolose,
  3                ##+ in modo particolare se usate con i caratteri
jolly.
  4
  5 #+ Continuazione di riga.
```

```
  6 #  Questa è la riga 1
  7 #+ di un commento posto su più righe,
  8 #+ e questa è la riga finale.
  9
 10 #* Nota.
 11
 12 #o Elemento di un elenco.
 13
 14 #> Alternativa.
 15 while [ "$var1" != "fine" ]    #> while test "$var1" != "fine"
```

- Un uso particolarmente intelligente dei costrutti if-test è quello per commentare blocchi di codice.

```
  1 #!/bin/bash
  2
  3 BLOCCO_DI_COMMENTO=
  4 #  Provate a impostare la variabile precedente ad un valore
qualsiasi
  5 #+ ed otterrete una spiacevole sorpresa.
  6
  7 if [ $BLOCCO_DI_COMMENTO ]; then
  8
  9 Commento --
 10 ========================================
 11 Questa è una riga di commento.
 12 Questa è un'altra riga di commento.
 13 Questa è un'altra riga ancora di commento.
 14 ========================================
 15
 16 echo "Questo messaggio non verrà visualizzato."
 17
 18 I blocchi di commento non generano errori! Wow!
 19
 20 fi
 21
 22 echo "Niente più commenti, prego."
 23
 24 exit 0
```

- Si confronti questo esempio con commentare un blocco di codice con gli here document.
- L'uso della variabile di exit status $? consente allo script di verificare se un parametro contiene solo delle cifre, così che possa essere trattato come un intero.

```
  1 #!/bin/bash
  2
  3 SUCCESSO=0
  4 E_ERR_INPUT=65
  5
  6 test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
  7 # Un intero è o uguale a 0 o diverso da 0.
  8 # 2>/dev/null sopprime il messaggio d'errore.
  9
 10 if [ $? -ne "$SUCCESSO" ]
 11 then
 12   echo "Utilizzo: `basename $0` intero"
 13   exit $E_ERR_INPUT
 14 fi
 15
```

```
 16 let "somma = $1 + 25"             #  Darebbe un errore se $1 non
 17                                   #+ fosse un intero.
 18 echo "Somma = $somma"
 19
 20 # In questo modo si può verificare qualsiasi variabile,
 21 #+ non solo i parametri passati da riga di comando.
 22
 23 exit 0
```

- L'intervallo 0 - 255, per i valori di ritorno di una funzione, rappresenta una seria limitazione. Anche l'impiego di variabili globali ed altri espedienti spesso si rivela problematico. Un metodo alternativo affinché la funzione restituisca un valore allo script, è fare in modo che questa scriva il "valore di ritorno" allo `stdout` (solitamente con echo) per poi assegnarlo a una variabile. In realtà si tratta di una variante della sostituzione di comando.

  Esempio 34-12. Uno stratagemma per il valore di ritorno

```
  1 #!/bin/bash
  2 # multiplication.sh
  3
  4 moltiplica ()                  # Moltiplica i parametri passati.
  5 {                              # Accetta un numero variabile di
argomenti.
  6
  7   local prodotto=1
  8
  9   until [ -z "$1" ]            # Finché ci sono parametri...
 10   do
 11     let "prodotto *= $1"
 12     shift
 13   done
 14
 15   echo $prodotto               #  Lo visualizza allo stdout,
 16 }                              #+ poiché verrà assegnato ad una
variabile.
 17
 18 molt1=15383; molt2=25211
 19 val1=`moltiplica $molt1 $molt2`
 20 echo "$molt1 X $molt2 = $val1"
 21                                # 387820813
 22
 23 molt1=25; molt2=5; molt3=20
 24 val2=`moltiplica $molt1 $molt2 $molt3`
 25 echo "$molt1 X $molt2 X $molt3 = $val2"
 26                                # 2500
 27
 28 molt1=188; molt2=37; molt3=25; molt4=47
 29 val3=`moltiplica $molt1 $molt2 $molt3 $molt4`
 30 echo "$molt1 X $molt2 X $molt3 X $molt4 = $val3"
 31                                # 8173300
 32
 33 exit 0
```

  La stessa tecnica funziona anche per le stringhe alfanumeriche. Questo significa che una funzione può "restituire" un valore non numerico.

```
  1 car_maiuscolo ()             #  Cambia in maiuscolo il carattere
iniziale
```

```
  2 {                           #+ di un argomento stringa/he passato.
  3
  4   stringa0="$@"             # Accetta più argomenti.
  5
  6   primocar=${stringa0:0:1}  # Primo carattere.
  7   stringa1=${stringa0:1}    # Parte restante della/e stringa/he.
  8
  9   PrimoCar=`echo "$primocar" | tr a-z A-Z`
 10                             # Cambia in maiuscolo il primo
carattere.
 11
 12   echo "$PrimoCar$stringa1"  # Visualizza allo stdout.
 13
 14 }
 15
 16 nuovastringa=`car_maiuscolo "ogni frase dovrebbe iniziare \
 17 con una lettera maiuscola."`
 18 echo "$nuovastringa"       #  Ogni frase dovrebbe iniziare con una
 19                           #+ lettera maiuscola.
```

Con questo sistema una funzione può "restituire" più valori.

Esempio 34-13. Un ulteriore stratagemma per il valore di "ritorno"

```
  1 #!/bin/bash
  2 # sum-product.sh
  3 # Una funzione può "restituire" più di un valore.
  4
  5 somma_e_prodotto ()   #  Calcola sia la somma che il prodotto degli
  6                       #+ argomenti passati.
  7 {
  8   echo $(( $1 + $2 )) $(( $1 * $2 ))
  9 # Visualizza allo stdout ogni valore calcolato, separato da uno
spazio.
 10 }
 11
 12 echo
 13 echo "Inserisci il primo numero"
 14 read primo
 15
 16 echo
 17 echo "Inserisci il secondo numero"
 18 read secondo
 19 echo
 20
 21 valres=`somma_e_prodotto $primo $secondo`    # Assegna l'output
della funzione.
 22 somma=`echo "$valres" | awk '{print $1}'`    # Assegna il primo
campo.
 23 prodotto=`echo "$valres" | awk '{print $2}'` # Assegna il secondo
campo.
 24
 25 echo "$primo + $secondo = $somma"
 26 echo "$primo * $secondo = $prodotto"
 27 echo
 28
 29 exit 0
```

- Le prossime, della serie di trucchi del mestiere, sono le tecniche per il passaggio di un array a una funzione e della successiva "restituzione" dell'array allo script.

Passare un array ad una funzione implica dover caricare gli elementi dell'array, separati da spazi, in una variabile per mezzo della sostituzione di comando. Per la restituzione dell'array, come "valore di ritorno" della funzione, si impiega lo stratagemma *appena descritto* e, quindi, tramite la sostituzione di comando e l'operatore **(...)** lo si riassegna ad un array.

Esempio 34-14. Passaggio e restituzione di array

```
 1 #!/bin/bash
 2 # array-function.sh: Passaggio di un array a una funzione e...
 3 #                   "restituzione" di un array da una funzione
 4
 5
 6 Passa_Array ()
 7 {
 8   local array_passato   # Variabile locale.
 9   array_passato=( `echo "$1"` )
10   echo "${array_passato[@]}"
11   #  Elenca tutti gli elementi del nuovo array
12   #+ dichiarato e impostato all'interno della funzione.
13 }
14
15
16 array_originario=( elemento1 elemento2 elemento3 elemento4 elemento5
)
17
18 echo
19 echo "array originario = ${array_originario[@]}"
20 #                        Elenca tutti gli elementi dell'array
originario.
21
22
23 # Ecco il trucco che consente di passare un array ad una funzione.
24 # ************************************
25 argomento=`echo ${array_originario[@]}`
26 # ************************************
27 #  Imposta la variabile
28 #+ a tutti gli elementi, separati da spazi, dell'array originario.
29 #
30 # È da notare che cercare di passare semplicemente l'array non
funziona.
31
32
33 # Ed ecco il trucco che permette di ottenere un array come "valore
di ritorno".
34 # ******************************************
35 array_restituito=( `Passa_Array "$argomento"` )
36 # ******************************************
37 # Assegna l'output 'visualizzato' della funzione all'array.
38
39 echo "array restituito = ${array_restituito[@]}"
40
41 echo "============================================================="
42
43 #  Ora, altra prova, un tentativo di accesso all'array (per
elencarne
44 #+ gli elementi) dall'esterno della funzione.
45 Passa_Array "$argomento"
46
47 # La funzione, di per sé, elenca l'array, ma...
```

```
48 #+ non è consentito accedere all'array al di fuori della funzione.
49 echo "Array passato (nella funzione) = ${array_passato[@]}"
50 # VALORE NULL perché è una variabile locale alla funzione.
51
52 echo
53
54 exit 0
```

Per un dimostrazione più elaborata di passaggio di array a funzioni, vedi Esempio A-11.

- Utilizzando il costrutto doppie parentesi è possibile l'impiego della sintassi in stile C per impostare ed incrementare le variabili, e per i cicli for e while. Vedi Esempio 10-12 e Esempio 10-17.
- Impostare path e umask all'inizio di uno script lo rende maggiormente "portabile" -- più probabilità che possa essere eseguito su una macchina "forestiera" il cui utente potrebbe aver combinato dei pasticci con `$PATH` e **umask**.

```
 1 #!/bin/bash
 2 PATH=/bin:/usr/bin:/usr/local/bin ; export PATH
 3 umask 022
 4
 5 # Grazie a Ian D. Allen per il suggerimento.
```

- Un'utile tecnica di scripting è quella di fornire *ripetitivamente* l'output di un filtro (con una pipe) allo *stesso filtro*, ma con una serie diversa di argomenti e/o di opzioni. tr e grep sono particolarmente adatti a questo scopo.

```
 1 Dall'esempio "wstrings.sh".
 2
 3 wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
 4 tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

- **Esempio 34-15. Divertirsi con gli anagrammi**

```
 1 #!/bin/bash
 2 # agram.sh: Giocare con gli anagrammi.
 3
 4 # Trova gli anagrammi di...
 5 LETTERE=etaoinshrdlu
 6
 7 anagram "$LETTERE" |   # Trova tutti gli anagrammi delle lettere
fornite...
 8 grep '.......' |       # Di almeno 7 lettere,
 9 grep '^is' |           # che iniziano con 'is'
10 grep -v 's$' |         # nessun plurale (in inglese, ovviamente
[N.d.T.])
11 grep -v 'ed$'          # nessun participio passato di verbi (come
sopra)
12 # E' possibile aggiungere molte altre combinazioni di condizioni
13
14 #  Usa l'utility "anagram" che fa parte del pacchetto
15 #+ dizionario "yawl" dell'autore di questo documento.
16 #  http://ibiblio.org/pub/Linux/libs/yawl-0.3.tar.gz
17 #  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz
18
19
20 exit 0                # Fine del codice.
```

```
21
22 bash$ sh agram.sh
23 islander
24 isolate
25 isolead
26 isotheral
```

- Vedi anche Esempio 28-3, Esempio 12-21 e Esempio A-10.
- Si usino gli "here document anonimi" per commentare blocchi di codice ed evitare di dover commentare ogni singola riga con un #. Vedi Esempio 17-11.
- Eseguire uno script su una macchina sulla quale non è installato il comando su cui lo script si basa, è pericoloso. Si usi whatis per evitare potenziali problemi.

```
 1 CMD=comando1                    # Scelta primaria.
 2 PianoB=comando2                 # Comando di ripiego.
 3
 4 verifica_comando=$(whatis "$CMD" | grep 'nothing appropriate')*
 5 #  Se 'comando1' non viene trovato sul sistema , 'whatis'
restituisce
 6 #+ "comando1: nothing appropriate."
 7 #
 8 #  Un'alternativa più sicura sarebbe:
 9 #     verifica_comando=$(whereis "$CMD" | grep \/)
10 #  Ma allora il senso della verifica seguente andrebbe invertito,
11 #+ dal momento che la variabile $verifica_comando è impostata solo
se
12 #+ $CMD è presente sul sistema.
13 #     (Grazie, bojster.)
14
15
16 if [[ -z "$verifica_comando" ]] # Verifica se il comando è presente.
17 then
18   $CMD opzione1 opzione2        #  Esegue comando1 con le opzioni.
19 else                           #  Altrimenti,
20   $PianoB                       #+ esegue comando2.
21 fi
22
23 #* Ma anche "niente di appropriato".
24 #  Verificatelo per la vostra distribuzione [N.d.T.]
```

- Una verifica if-grep potrebbe non dare i risultati attesi in caso di errore, quando il testo viene visualizzato allo `stderr` invece che allo `stdout`.

```
 1 if ls -l file_inesistente | grep -q 'No such file or directory'
 2   then echo "Il file \"file_inesistente\" non esiste."
 3 fi
```

- Il problema può essere risolto con la redirezione dello `stderr` allo `stdout`.

```
 1 if ls -l file_inesistente 2>&1 | grep -q 'No such file or directory'
 2 #                          ^^^^
 3   then echo "Il file \"file_inesistente\" non esiste."
 4 fi
 5
 6 # Grazie a Chris Martin per la precisazione.
```

- Il comando run-parts è utile per eseguire una serie di comandi in sequenza, in particolare abbinato a cron o at.
- Sarebbe bello poter invocare i widget X-Windows da uno script di shell. Si dà il caso che esistano diversi pacchetti che hanno la pretesa di far questo, in particolare *Xscript*, *Xmenu* e *widtools*. Sembra, però, che i primi due non siano più mantenuti. Fortunatamente è ancora possibile ottenere *widtools* qui.

  
  Il pacchetto *widtools* (widget tools) richiede l'installazione della libreria *XForms*. In aggiunta, il Makefile necessita di alcune sistemazioni prima che il pacchetto possa essere compilato su un tipico sistema Linux. Infine, tre dei sei widget non funzionano (segmentation fault).

- La famiglia di strumenti *dialog* offre un metodo per richiamare i widget di "dialogo" da uno script di shell. L'utility originale **dialog** funziona in una console di testo, mentre i suoi successori **gdialog**, **Xdialog** e **kdialog** usano serie di widget basate su X-Windows.
- **Esempio 34-16. Widget invocati da uno script di shell**

```
  1 #!/bin/bash
  2 # dialog.sh: Uso dei widgets 'gdialog'.
  3 # Per l'esecuzione dello script è indispensabile aver installato
'gdialog'.
  4
  5 # Lo script è stato ispirato dal seguente articolo.
  6 #     "Scripting for X Productivity," di Marco Fioretti,
  7 #      LINUX JOURNAL, Numero 113, Settembre 2003, pp. 86-9.
  8 # Grazie a tutti quelli di LJ.
  9
 10
 11 # Errore di input nel box di dialogo.
 12 E_INPUT=65
 13 # Dimensioni dei widgets di visualizzazione e di input.
 14 ALTEZZA=50
 15 LARGHEZZA=60
 16
 17 # Nome del file di output (composto con il nome dello script).
 18 OUTFILE=$0.output
 19
 20 # Visualizza questo script in un widget di testo.
 21 gdialog --title "Visualizzazione: $0" --textbox $0 $ALTEZZA
$LARGHEZZA
 22
 23
 24 # Ora, proviamo a salvare l'input in un file.
 25 echo -n "VARIABILE=\"" > $OUTFILE  # Usate il quoting nel caso
l'input
 26                                    #+ contenga degli spazi.
 27 gdialog --title "Input Utente" --inputbox "Prego, inserisci un
dato:" \
 28 $ALTEZZA $LARGHEZZA 2>> $OUTFILE
 29
 30
 31 if [ "$?" -eq 0 ]
 32 # È buona pratica controllare l'exit status.
 33 then
 34   echo "Eseguito \"box di dialogo\" senza errori."
 35 else
 36   echo "Errore(i) nell'esecuzione di \"box di dialogo\"."
 37         # Oppure avete cliccato su "Cancel" invece che su "OK".
```

```
 38   rm $OUTFILE
 39   exit $E_INPUT
 40 fi
 41
 42
 43 echo -n "\"" >> $OUTFILE           # Virgolette finali alla
variabile.
 44 #  Questo comando è stato posto qui in fondo per non confondere
 45 #+ l'exit status precedente.
 46
 47
 48 # Ora, recuperiamo e visualizziamo la variabile.
 49 . $OUTFILE   # 'Include' il file salvato.
 50 echo "La variabile inserita nel \"box di input\" è: "$VARIABILE""
 51
 52 rm $OUTFILE  #  Cancellazione del file temporaneo.
 53              #  Alcune applicazioni potrebbero aver ancora bisogno
 54              #+ di questo file.
 55 exit 0
```

- Per altri metodi di scripting con l'impiego di widget, si provino *Tk* o *wish* (derivati *Tcl*), *PerlTk* (Perl con estensioni Tk), *tksh* (ksh con estensioni Tk), *XForms4Perl* (Perl con estensioni XForms), *Gtk-Perl* (Perl con estensioni Gtk) o *PyQt* (Python con estensioni Qt).
- Per effettuare revisioni multiple di uno script complesso, si usi il pacchetto *rcs* Revision Control System.

  Tra le sue funzionalità vi è anche quella di aggiornare automaticamente l'ID dell'intestazione. Il comando **co** di *rcs* effettua una sostituzione di parametro di alcune parole chiave riservate ad esempio, rimpiazza *`#$Id$`* di uno script con qualcosa come:

```
  1 #$Id: hello-world.sh,v 1.1 2004/10/16 02:43:05 bozo Exp $
```

34.8. Sicurezza

A questo punto è opportuno un breve avvertimento sulla sicurezza degli script. Uno script di shell può contenere un *worm*, un *trojan* o persino un *virus*. Per questo motivo, non bisogna mai eseguire uno script da root (o consentire che sia inserito tra gli script di avvio del sistema in `/etc/rc.d`), a meno che non si sia ottenuto tale script da una fonte fidata o non lo si sia analizzato attentamente per essere sicuri che non faccia niente di dannoso.

Diversi ricercatori dei Bell Labs, e di altri istituti, tra i quali M. Douglas McIlroy, Tom Duff e Fred Cohen, che hanno indagato le implicazioni dei virus negli script di shell, sono giunti alla conclusione che è fin troppo facile, anche per un principiante, uno "script kiddie", scriverne uno. [1]

Questa è un'altra ragione ancora per imparare lo scripting. Essere in grado di visionare e capire gli script è un mezzo per proteggere il sistema da danni o dall'hacking.

Note

[1] Vedi l'articolo di Marius van Oers, Unix Shell Scripting Malware e anche *Denning* in bibliografia.

34.9. Portabilità

Questo libro tratta specificamente dello scripting di Bash su un sistema GNU/Linux. Nondimeno, gli utilizzatori di **sh** e **ksh** vi troveranno molti utili argomenti.

Attualmente, molte delle diverse shell e linguaggi di scripting tendono ad uniformarsi allo standard POSIX 1003.2. Invocare Bash con l'opzione `--posix`, o inserire nello script l'intestazione **set -o posix**, fa sì che Bash si conformi in maniera molto stretta a questo standard. Un'altra alternativa è usare nello script l'intestazione

```
1 #!/bin/sh
```

al posto di

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

Va notato che `/bin/sh` è un link a `/bin/bash` in Linux e in alcune altre versioni di UNIX, e che uno script così invocato disabilita le funzionalità estese di Bash.

La maggior parte degli script Bash funzionano senza alcuna modifica con **ksh**, e viceversa, perché Chet Ramey sta alacremente adattando per Bash, nelle sue più recenti versioni, le funzionalità di **ksh**.

Su una macchina commerciale UNIX, gli script che utilizzano le funzionalità specifiche GNU dei comandi standard potrebbero non funzionare. Negli ultimi anni questo è diventato un problema meno rilevante, dal momento che le utility GNU hanno rimpiazzato una parte piuttosto consistente delle analoghe controparti proprietarie, persino sui "grandi cervelloni" UNIX. Il rilascio, da parte di Caldera, dei codici sorgente di molte delle utility originali UNIX ha accelerato questa tendenza.

Bash possiede alcune funzionalità non presenti nella tradizionale shell Bourne. Tra le altre:

- Alcune opzioni d'invocazione estese
- La sostituzione di comando con la notazione **$( )**
- Alcune operazioni di manipolazione di stringa
- La sostituzione di processo
- I builtin specifici di Bash

Vedi Bash F.A.Q. per un elenco completo.

34.10. Lo scripting di shell sotto Windows

Anche gli utilizzatori di *quell'altro* SO possono eseguire script di shell in stile UNIX e, quindi, beneficiare di molte delle lezioni di questo libro. Il pacchetto Cygwin, di Cygnus, e le MKS utilities, di Mortice Kern Associates, aggiungono a Windows le capacità dello scripting di shell.

Circolano delle voci su una futura versione di Windows contenente funzionalità di scripting da riga di comando simili a Bash, ma questo resta ancora tutto da vedere.

Capitolo 35. Bash, versioni 2 e 3

Sommario

35.1. Bash, versione 2

La versione corrente di *Bash*, quella che viene eseguita sulla vostra macchina, attualmente è la 2.XX.Y o la 3.xx.y..

```
bash$ echo $BASH_VERSION
2.05.b.0(1)-release
```

La versione 2, aggiornamento del classico linguaggio di scripting di Bash, ha aggiunto, gli array, [1] l'espansione di stringa e di parametro, e un metodo migliore per le referenziazioni indirette a variabili.

Esempio 35-1. Espansione di stringa

```
 1 #!/bin/bash
 2
 3 # Espansione di stringa.
 4 # Introdotta con la versione 2 di Bash.
 5
 6 #  Le stringhe nella forma $'xxx'
 7 #+ consentono l'interpretazione delle sequenze di escape standard.
 8
 9 echo $'Tre segnali acustici \a \a \a'
10      # Su alcuni terminali potrebbe venir eseguito un solo segnale acustico.
11 echo $'Tre form feed \f \f \f'
12 echo $'10 ritorni a capo \n\n\n\n\n\n\n\n\n\n'
13 echo $'\102\141\163\150'   # Bash
14                           # Valori ottali di ciascun carattere.
15
16 exit 0
```

Esempio 35-2. Referenziazioni indirette a variabili - una forma nuova

```
 1 #!/bin/bash
 2
 3 # Referenziazione indiretta a variabile.
 4 # Possiede alcuni degli attributi delle referenziazioni del C++.
 5
 6
 7 a=lettera_alfabetica
 8 lettera_alfabetica=z
 9
10 echo "a = $a"           # Referenziazione diretta.
11
12 echo "Ora a = ${!a}"    # Referenziazione indiretta.
13 #  La notazione ${!variabile} è di molto superiore alla vecchia
14 #+ "eval var1=\$$var2"
```

```
15
16 echo
17
18 t=cella_3
19 cella_3=24
20 echo "t = ${!t}"                              # t = 24
21 cella_3=387
22 echo "Il valore di t è cambiato in ${!t}"    # 387
23
24 #  È utile per il riferimento ai membri di un array o di una tabella,
25 #+ o per simulare un array multidimensionale.
26 #  Un'opzione d'indicizzazione sarebbe stata più gradita (sigh).
27
28 exit 0
```

Esempio 35-3. Applicazione di un semplice database, con l'utilizzo della referenziazione indiretta

```
 1 #!/bin/bash
 2 #  resistor-inventory.sh
 3 #  Applicazione di un semplice database che utilizza la referenziazione
 4 #+ indiretta alle variabili.
 5
 6 # ============================================================== #
 7 # Dati
 8
 9 B1723_valore=470                               # Ohm
10 B1723_potenzadissip=.25                        # Watt
11 B1723_colori="giallo-viola-marrone"            # Colori di codice
12 B1723_loc=173                                  # Posizione
13 B1723_inventario=78                            # Quantità
14
15 B1724_valore=1000
16 B1724_potenzadissip=.25
17 B1724_colori="marrone-nero-rosso"
18 B1724_loc=24N
19 B1724_inventario=243
20
21 B1725_valore=10000
22 B1725_potenzadissip=.25
23 B1725_colori="marrone-nero-arancione"
24 B1725_loc=24N
25 B1725_inventario=89
26
27 # ============================================================== #
28
29
30 echo
31
32 PS3='Inserisci il numero di catalogo: '
33
34 echo
35
36 select numero_catalogo in "B1723" "B1724" "B1725"
37 do
38   Inv=${numero_catalogo}_inventario
39   Val=${numero_catalogo}_valore
40   Pdissip=${numero_catalogo}_potenzadissip
41   Loc=${numero_catalogo}_loc
42   Codcol=${numero_catalogo}_colori
43
```

```
44   echo
45   echo "Numero di catalogo $numero_catalogo:"
46   echo "In magazzino ci sono ${!Inv} resistori da\
47 [${!Val} ohm / ${!Pdissip} watt]."
48   echo "Si trovano nel contenitore nr. ${!Loc}."
49   echo "Il loro colore di codice è \"${!Codcol}\"."
50
51   break
52 done
53
54 echo; echo
55
56 # Esercizi:
57 # --------
58 # 1) Riscrivete lo script in modo che legga i dati da un file esterno.
59 # 2) Riscrivete lo script utilizzando gli array, al posto della
60 #+   referenziazione indiretta a variabile.
61 #    Quale, tra i due, è il metodo più diretto e intuitivo?
62
63
64 # Nota:
65 # -----
66 #  Gli script di shell non sono appropriati per le applicazioni di database,
67 #+ tranne quelle più semplici. Anche in questi casi, però,
68 #+ bisogna ricorrere ad espedienti e trucchi vari.
69 #  È molto meglio utilizzare un linguaggio che abbia un
70 #+ supporto nativo per le strutture, come C++ o Java (o anche Perl).
71
72 exit 0
```

Esempio 35-4. Utilizzo degli array e di vari altri espedienti per simulare la distribuzione casuale di un mazzo di carte a 4 giocatori

```
 1 #!/bin/bash
 2 #  Su macchine un po' datate, potrebbe essere necessario invocarlo
 3 #+ con  #!/bin/bash2.
 4
 5 # Carte:
 6 # Distribuzione di un mazzo di carte a quattro giocatori.
 7
 8 NONDISTRIBUITA=0
 9 DISTRIBUITA=1
10
11 GIÀ_ASSEGNATA=99
12
13 LIMITE_INFERIORE=0
14 LIMITE_SUPERIORE=51
15 CARTE_PER_SEME=13
16 CARTE=52
17
18 declare -a Mazzo
19 declare -a Semi
20 declare -a Carte
21 #  Sarebbe stato più semplice ed intuitivo
22 #+ con un unico array tridimensionale.
23 #  Forse una futura versione di Bash supporterà gli array multidimensionali.
24
25
26 Inizializza_Mazzo ()
27 {
28 i=$LIMITE_INFERIORE
```

```
until [ "$i" -gt $LIMITE_SUPERIORE ]
do
  Mazzo[i]=$NONDISTRIBUITA   #  Imposta ogni carta del "Mazzo" come non
                             #+ distribuita.
  let "i += 1"
done
echo
}

Inizializza_Semi ()
{
Semi[0]=F #Fiori
Semi[1]=Q #Quadri
Semi[2]=C #Cuori
Semi[3]=P #Picche
}

Inizializza_Carte ()
{
Carte=(2 3 4 5 6 7 8 9 10 J Q K A)
# Metodo alternativo di inizializzazione di array.
}

Sceglie_Carta ()
{
numero_carta=$RANDOM
let "numero_carta %= $CARTE"
if [ "${Mazzo[numero_carta]}" -eq $NONDISTRIBUITA ]
then
  Mazzo[numero_carta]=$DISTRIBUITA
  return $numero_carta
else
  return $GIÀ_ASSEGNATA
fi
}

Determina_Carta ()
{
numero=$1
let "numero_seme = numero / CARTE_PER_SEME"
seme=${Semi[numero_seme]}
echo -n "$seme-"
let "nr_carta = numero % CARTE_PER_SEME"
Carta=${Carte[nr_carta]}
printf %-4s $Carta
# Visualizza le carte ben ordinate per colonne.
}

Seme_Casuale ()  # Imposta il seme del generatore di numeri casuali.
{                # Cosa succederebbe se questo non venisse fatto?
Seme=`eval date +%s`
let "Seme %= 32766"
RANDOM=$Seme
}

Da_Carte ()
{
echo

carte_date=0
while [ "$carte_date" -le $LIMITE_SUPERIORE ]
do
```

```
 91   Sceglie_Carta
 92   t=$?
 93
 94   if [ "$t" -ne $GIÀ_ASSEGNATA ]
 95   then
 96     Determina_Carta $t
 97
 98     u=$carte_date+1
 99     # Ritorniamo all'indicizzazione in base 1 (temporaneamente). Perchè?
100     let "u %= $CARTE_PER_SEME"
101     if [ "$u" -eq 0 ]   # Costrutto condizionale if/then annidato.
102     then
103      echo
104      echo
105     fi
106     # Separa i giocatori.
107
108     let "carte_date += 1"
109   fi
110 done
111
112 echo
113
114 return 0
115 }
116
117
118 # Programmazione strutturata:
119 # l'intero programma è stato "modularizzato" per mezzo delle Funzioni.
120
121 #================
122 Seme_Casuale
123 Inizializza_Mazzo
124 Inizializza_Semi
125 Inizializza_Carte
126 Da_Carte
127 #================
128
129 exit 0
130
131
132
133
134 #  Esercizio 1:
135 #  Aggiungete commenti che spieghino completamente lo script.
136
137 #  Esercizio 2:
138 #  Aggiungete una routine (funzione) per visualizzare la distribuzione
ordinata
139 #+ per seme.
140 #  Potete aggiungere altri fronzoli, si vi aggrada.
141
142 #  Esercizio 3:
143 #  Semplificate e raffinate la logica dello script.
```

Note

[1] Chet Ramey ha promesso gli array associativi (una funzionalità Perl) in una futura release di Bash. Questo non è ancora avvenuto, neanche nella versione 3.

35.2. Bash, versione 3

Il 27 luglio 2004, Chet Ramey ha rilasciato la versione 3 di Bash. Questo aggiornamento corregge un certo numero di errori presenti in Bash e aggiunge alcune nuove funzionalità.

Eccone alcune:

- Un nuovo, più generale, operatore per l'espansione sequenziale **{a..z}**.

```
 1 #!/bin/bash
 2
 3 for i in {1..10}
 4 #  Più semplice e più diretto di
 5 #+ for i in $(seq 10)
 6 do
 7   echo -n "$i "
 8 done
 9
10 echo
11
12 # 1 2 3 4 5 6 7 8 9 10
```

- L'operatore **${!array[@]}**, che espande a tutti gli indici di un dato array.

```
 1 #!/bin/bash
 2
 3 Array=(elemento-zero elemento-uno elemento-due elemento-tre)
 4
 5 echo ${Array[0]}   # elemento-zero
 6                    # Primo elemento dell'array.
 7
 8 echo ${!Array[@]}  # 0 1 2 3
 9                    # Tutti gli indici di Array.
10
11 for i in ${!Array[@]}
12 do
13   echo ${Array[i]} # elemento-zero
14                    # elemento-uno
15                    # elemento-due
16                    # elemento-tre
17                    #
18                    # Tutti gli elementi di Array.
19 done
```

- L'operatore di ricerca di corrispondenza =~ delle Espressioni Regolari all'interno del costrutto di verifica doppie parentesi quadre. (Perl possiede un operatore simile.)

```
 1 #!/bin/bash
 2
 3 variabile="Questo è un bel pasticcio."
 4
 5 echo "$variabile"
 6
 7 if [[ "$variabile" =~ "Q*bel*ccio*" ]]
 8 #  Ricerca di corrispondenza "Regex" con l'operatore =~
 9 #+ inserito tra [[ doppie parentesi quadre ]].
10 then
```

```
11   echo "trovata corrispondenza"
12       # trovata corrispondenza
13 fi
```

Capitolo 36. Note conclusive

Sommario

36.1. Nota dell'autore

doce ut discas

(Teach, that you yourself may learn.)

Come sono arrivato a scrivere un libro sullo scripting di Bash? È una strana storia. È capitato che un paio d'anni fa avessi bisogno di imparare lo scripting di shell -- e quale modo migliore per farlo se non leggere un buon libro sul tema? Mi misi a cercare un manuale introduttivo e una guida di riferimento che trattassero ogni aspetto dell'argomento. Cercavo un libro che avrebbe dovuto cogliere i concetti difficili, sviscerarli e spiegarli dettagliatamente per mezzo di esempi ben commentati. [1] In effetti, stavo cercando *proprio questo libro*, o qualcosa di molto simile. Purtroppo, un tale libro non esisteva e, se l'avessi voluto, avrei dovuto scriverlo. E quindi, eccoci qua.

Questo fatto mi ricorda la storia, non vera, del professore pazzo. Il tizio era matto come un cavallo. Alla vista di un libro, uno qualsiasi -- in biblioteca, in una libreria, ovunque -- diventava ossessionato dall'idea che egli stesso avrebbe potuto scriverlo, avrebbe dovuto scriverlo e avrebbe fatto, per di più, un lavoro migliore. Al che, si precipitava a casa e procedeva nel suo intento, scrivere un libro con lo stesso, identico titolo. Alla sua morte, qualche anno più tardi, presumibilmente avrà avuto a suo credito migliaia di libri, roba da far vergognare persino Asimov. I libri, forse, avrebbero potuto anche non essere dei buoni libri -- chi può saperlo -- ma è questo quello che conta veramente? Ecco una persona che ha vissuto il suo sogno, sebbene ne fosse ossessionato e da esso sospinto. Ed io non posso fare a meno di ammirare quel vecchio sciocco...

Note

[1] Trattasi della celebre tecnica dello "spremere come un limone".

36.2. A proposito dell'autore

Ad ogni modo, chi è costui?

L'autore non rivendica particolari credenziali o qualifiche, tranne il bisogno di scrivere. [1] Questo libro è un po' il punto di partenza per l'altro suo maggior lavoro, HOW-2 Meet Women: The Shy Man's Guide to Relationships. Ha inoltre scritto Software-Building HOWTO. In seguito, si è cimentato per la prima volta in una fiction breve.

Utente Linux dal 1995 (Slackware 2.2, kernel 1.2.1), l'autore ha rilasciato alcuni programmini, tra i quali cruft, utility di cifratura one-time pad; mcalc, per il calcolo del piano d'ammortamento di un mutuo; judge, arbitro per le partite di Scrabble® e il pacchetto yawl per giochi di parole. Ha iniziato a programmare usando il FORTRAN IV su CDC 3800, ma non ha neanche un po' di nostalgia di quei giorni.

Vivendo, con la moglie e il cane, presso una solitaria comunità nel deserto, riconosce il valore della fragilità umana.

Note

[1] Chi può, fa. Chi non può... prende un MCSE (Microsoft Certified Systems Engineer - Attestato di Tecnico di Sistemi Certificato Microsoft [N.d.T.]).

36.3. Dove cercare aiuto

L'autore di solito, se non troppo occupato (e nel giusto stato d'animo), risponde su questioni riguardanti lo scripting in generale. Tuttavia, nel caso di un problema riguardante il funzionamento di uno script particolare, si consiglia vivamente di inviare una richiesta al newsgroup Usenet comp.os.unix.shell.

36.4. Strumenti utilizzati per produrre questo libro

36.4.1. Hardware

Un portatile usato IBM Thinkpad, modello 760X (P166, 104 mega RAM) con Red Hat 7.1/7.3. Certo, è lento ed ha una tastiera strana, ma è sempre più veloce di un Bloc Notes e di una matita N. 2.

36.4.2. Software e Printware

i. Il potente editor di testi vim di Bram Moolenaar, in modalità SGML.
ii. OpenJade, motore di rendering DSSSL, per la conversione di documenti SGML in altri formati.
iii. I fogli di stile DSSSL di Norman Walsh.
iv. *DocBook, The Definitive Guide*, di Norman Walsh e Leonard Muellner (O'Reilly, ISBN 1-56592-580-7). È la guida di riferimento standard per tutti coloro che vogliono scrivere un documento in formato Docbook SGML.

36.5. Ringraziamenti

Questo progetto è stato reso possibile dalla partecipazione collettiva. L'autore riconosce, con gratitudine, che sarebbe stato un compito impossibile scrivere questo libro senza l'aiuto ed il riscontro di tutte le persone elencate di seguito.

Philippe Martin ha tradotto questo documento in formato DocBook/SGML. Quando non impegnato come sviluppatore software in una piccola società francese, si diletta lavorando sul software e sulla documentazione GNU/Linux, leggendo, suonando e, per la pace del suo spirito, facendo baldoria con gli amici. Potreste incrociarlo da qualche parte, in Francia o nei paesi baschi, o inviandogli un email a feloy@free.fr.

Philippe Martin ha evidenziato, tra l'altro, che sono possibili i parametri posizionali oltre $9 per mezzo della notazione {parentesi graffe}, vedi Esempio 4-5.

Stephane Chazelas ha fornito un lungo elenco di correzioni, aggiunte e script d'esempio. Più che un collaboratore, ha assunto, in effetti, il ruolo di **curatore** di questo documento. Merci beaucoup!

Paulo Marcel Coelho Aragao per le molte correzioni, importanti o meno, e per aver fornito un buon numero di utili suggerimenti.

Vorrei ringraziare in particolare *Patrick Callahan*, *Mike Novak* e *Pal Domokos* per aver scovato errori, sottolineato ambiguità, e per aver suggerito chiarimenti e modifiche. La loro vivace discussione sullo scripting di shell e sulle questioni generali inerenti alla documentazione, mi hanno indotto a cercare di rendere più interessante questo documento.

Sono grato a Jim Van Zandt per aver evidenziato errori e omissioni nella versione 0.2 di questo documento. Ha fornito anche un istruttivo script d'esempio.

Molte grazie a Jordi Sanfeliu, per aver concesso il permesso all'uso del suo bello script tree (Esempio A-18), e a Rick Boivie, per averlo revisionato.

Allo stesso modo, grazie a Michel Charpentier per il consenso all'uso del suo script per la fattorizzazione dc (Esempio 12-43).

Onore a Noah Friedman per aver permesso l'utilizzo del suo script di funzioni stringa (Esempio A-19).

Emmanuel Rouat ha suggerito correzioni ed aggiunte sulla sostituzione di comando e sugli alias. Ha anche fornito un esempio molto bello di file `.bashrc` (Appendice J).

Heiner Steven ha gentilmente acconsentito all'uso del suo script per la conversione di base, Esempio 12-39. Ha, inoltre, eseguito numerose correzioni e fornito utili suggerimenti. Un grazie particolare.

Rick Boivie ha fornito il delizioso script ricorsivo *pb.sh* (Esempio 34-7), revisionato lo script *tree.sh* (Esempio A-18) e suggerito miglioramenti per le prestazioni dello script *monthlypmt.sh* (Esempio 12-38).

Florian Wisser mi ha chiarito alcune sfumature della verifica delle stringhe (vedi Esempio 7-6) ed altri argomenti.

Oleg Philon ha fornito suggerimenti riguardanti cut e pidof.

Michael Zick ha esteso l'esempio dell'array vuoto per dimostrare alcune sorprendenti proprietà degli array. Ha fornito anche altri esempi riguardanti questo argomento.

Marc-Jano Knopp ha segnalato correzioni sui file batch DOS.

Hyun Jin Cha ha trovato diversi errori tipografici durante la tarduzione in coreano del documento. Grazie per averli evidenziati.

Andreas Abraham ha inviato un lungo elenco di errori tipografici ed altre correzioni. Un grazie particolare!

Altri che hanno fornito script, utili suggerimenti e puntualizzato errori sono Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage (idee per script!), Rich Bartell, Jess Thrysoee, Adam Lazur, Bram Moolenaar, Baris Cicek, Greg Keraunen, Keith Matthews, Sandro Magi, Albert Reiner, Dim Segebart, Rory Winston, Lee Bigelow, Wayne Pollock, "jipe", "Mark," "bojster", "Ender", Emilio Conti, Ian. D. Allen, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Björn Eriksson, "nyal," John MacDonald, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Jeremy Impson, Ken Fuchs, Frank Wang, Sylvain Fourmanoit, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Stefano Palmeri, Alfredo Pironti e David Lawyer (egli stesso autore di quattro HOWTO).

La mia gratitudine a Chet Ramey e Brian Fox per aver scritto **Bash**, uno strumento per lo scripting elegante e potente.

Un grazie molto particolare per il lavoro accurato e determinato dei volontari del Linux Documentation Project. LDP ospita una vasta collezione di sapere ed erudizione Linux ed ha, in larga misura, reso possibile la pubblicazione di questo libro.

Stima e ringraziamenti a IBM, Novell, Red Hat, la Free Software Foundation e a tutte quelle ottime persone che combattono la giusta battaglia per mantenere il software Open Source libero e aperto.

Grazie soprattutto a mia moglie, Anita, per il suo incoraggiamento e supporto emozionale

Guida a cura dello (Staff CasertaGLUG) manuale distribuibile secondo la licenza GNU.