Guida avanzata di scripting Bash

Un’approfondita esplorazione dell’arte dello scripting di shell

Mendel Cooper

thegrendel@theriver.com

Guida avanzata di scripting Bash: Un’approfondita esplorazione dell’arte dello scripting di shell
by Mendel Cooper

Published 03 gennaio 2004

Questo manuale, per la cui comprensione non è necessaria una precedente conoscenza di scripting o di programmazione, permette di raggiungere rapidamente un livello di apprendimento intermedio/avanzato . . . *tempo che tranquillamente ed inconsapevolmente si trasforma in piccoli frammenti di conoscenza e saggezza UNIX®*. Può essere utilizzato come libro di testo, come manuale per autoapprendimento e come guida di riferimento per le tecniche di scripting di shell. Gli esercizi e gli esempi ampiamente commentati coinvolgono il lettore interessato, con l’avvertenza che **`l’unico modo per imparare veramente lo scripting è quello di scrivere script`**.

Questo libro è adatto per l’insegnamento scolastico, come introduzione generale ai concetti della programmazione.

L’ultimo aggiornamento di questo documento, in forma compressa bzip2 “tarball” comprendente sia i sorgenti SGML che il formato HTML, può essere scaricato dal sito dell’autore (http://personal.riverusers.com/~thegrendel/abs-guide-2.3.tar.bz2). Vedi il change log (http://personal.riverusers.com/~thegrendel/Change.log) per la cronologia delle revisioni.

Per la versione in lingua italiana sarà possibile reperirne una copia presso il PLUTO (http://www.pluto.linux.it/ildp/guide.html), ovvero il sito italiano collegato a tldp.

Al momento di cominciare la traduzione della Advanced Bash-Scripting Guide decisi che avrei adottato la forma impersonale. Procedendo nella traduzione, mi sono poi accorto, soprattutto negli esercizi di esempio, che l’autore usava giochi di parole, modi di dire e battute che rendevano la forma impersonale piuttosto problematica. In conseguenza di ciò, ho mantenuto tale forma solo nella parte esplicativa del documento, mentre ho adottato la forma personale nella traduzione degli esercizi. Questa dicotomia potrebbe sembrare stridente, ed in effetti un po’ lo è. Per questo motivo mi sono riproposto, durante gli aggiornamenti alle versioni successive, di adottare la forma personale per tutto il documento in modo che risulti meno "accademico" e "pesante".

Non ho tradotto gli esercizi dell’Appendice A perché di fronte alla scelta tra il perdere ulteriore tempo nella traduzione di tecniche avanzate e l’"urgenza" di rendere disponibile il documento, ho optato per la seconda alternativa. Sarà comunque mia cura, una volta effettuati tutti gli aggiornamenti, colmare anche questa lacuna. Naturalmente, non sono stati tradotti gli avvisi di sistema.

Vorrei ringraziare tutti i volontari di ILDP che mi hanno aiutato nel lavoro con i loro consigli.

Un grazie particolare a Ferdinando Ferranti (mailto:zappagalattica@inwind.it) per il suo lavoro di revisore e per aver trasformato la versione italiana del documento dal formato HTML in formato SGML DocBook.

Emilio Conti (mailto:em.conti@tin.it)

Dedication

Per Anita, fonte di ogni magia

Table of Contents

List of Tables

List of Examples

Part 1. Introduzione

La shell è un interprete di comandi. Molto più che una semplice interfaccia tra il kernel del sistema operativo e l'utilizzatore, è anche un vero e proprio potente linguaggio di programmazione. Un programma di shell, chiamato *script*, è uno strumento semplice da usare per creare applicazioni "incollando" insieme chiamate di sistema, strumenti, utility e file binari (eseguibili). Uno script di shell può utilizzare virtualmente l'intero repertorio di comandi, utility e strumenti UNIX. Se ciò non fosse abbastanza, i comandi interni della shell, come i costrutti di verifica ed i cicli, forniscono ulteriore potenza e flessibilità agli script. Questi si prestano eccezionalmente bene a compiti di amministrazione di sistema e a lavori ripetitivi e di routine, senza l'enfasi di un complesso, e fortemente strutturato, linguaggio di programmazione.

Chapter 1. Perché programmare la shell?

La conoscenza pratica dello scripting di shell è essenziale per coloro che desiderano diventare degli esperti amministratori di sistema, anche se mai avevano messo in preventivo di scrivere degli script. Occorre considerare che quando viene avviata una macchina Linux, questa esegue gli script di shell contenuti nel file `/etc/rc.d` per ripristinare la configurazione del sistema ed attivare i servizi. La comprensione dettagliata degli script di avvio è importante per analizzare il comportamento di un sistema e, se possibile, modificarlo.

Imparare a scrivere degli script non è difficile, perché possono essere costituiti da sezioni di piccole dimensioni ed è veramente esigua la serie di operatori ed opzioni specifiche [1] che è necessario conoscere. La sintassi è semplice e chiara, come quella necessaria per eseguire e concatenare utility da riga di comando, e sono anche poche le “regole” da imparare. Nella maggior parte dei casi, gli script di piccole dimensioni funzionano correttamente fin dalla prima volta che vengono eseguiti e non è complicata neanche la fase di debugging di quelli di dimensioni maggiori.

Uno script di shell è un metodo “rapido e grezzo” per costruire un prototipo di un’applicazione complessa. Eseguire anche una serie ridotta di istruzioni tramite uno script di shell è spesso un utile primo passo nello sviluppo di un progetto. In questo modo si può verificare e sperimentare la struttura di un’applicazione e scoprire i principali errori prima di procedere alla codifica finale in C, C++, Java o Perl.

Lo scripting di shell è attento alla filosofia classica UNIX di suddividere progetti complessi in sezioni di minori dimensioni che svolgono un compito particolare, collegando componenti e utility. Questo è considerato, da molti, un approccio migliore, o almeno esteticamente più piacevole per risolvere un problema, che utilizzare uno dei linguaggi di nuova generazione , come Perl, che offrono funzionalità per ogni esigenza, ma al prezzo di costringere a modificare il modo di pensare un progetto per adattarlo al linguaggio utilizzato.

Quando non usare gli script di shell

- in compiti che richiedono un utilizzo intenso di risorse, specialmente quando la velocità è un fattore determinante (ordinamenti, hashing, etc.)
- in procedure che comprendono operazioni matematiche complesse, specialmente aritmetica in virgola mobile, calcoli in precisione arbitraria o numeri complessi (si usi C++ o FORTRAN)
- è necessaria la portabilità (si usi il C)
- in applicazioni complesse dove è necessaria la programmazione strutturata (necessità di tipizzazione delle variabili, prototipi di funzione, etc.)
- in applicazioni particolari su cui si sta rischiando il tutto per tutto, o il futuro della propria società
- in situazioni in cui la sicurezza è importante, dove occorre garantire l’integrità del sistema e proteggerlo contro intrusioni, cracking e vandalismi
- in progetti costituiti da sotto-componenti con dipendenze interconnesse
- sono richieste operazioni su file di grandi dimensioni (Bash si limita ad un accesso sequenziale ai file, eseguito riga per riga e in un modo particolarmente goffo ed inefficiente)
- sono necessari gli array multidimensionali
- sono necessarie strutture di dati quali le liste collegate o gli alberi
- è necessario generare o manipolare grafici o GUI
- è necessario un accesso diretto all’hardware del sistema
- è necessaria una porta o un socket I/O

- è necessario l'utilizzo di librerie o interfacce per l'esecuzione di vecchio codice
- in applicazioni proprietarie a codice chiuso (gli script di shell mantengono aperto il codice sorgente permettendo a tutti di esaminarlo)

Nel caso si sia di fronte ad una o più delle eventualità appena descritte, occorre prendere in considerazione un linguaggio di scripting più potente, che potrebbe essere Perl, Tcl, Python, Ruby, o la possibilità di un linguaggio compilato di alto livello, quale il C, C++ o Java. Anche in questo caso, però, eseguire dei prototipi di un'applicazione come script di shell potrebbe costituire un'utile base di sviluppo.

Sarà utilizzata Bash, acronimo di "Bourne-Again Shell", e un po' un gioco di parole sull'ormai classica Shell Bourne di Stephen Bourne. Bash è diventata uno standard *de facto* dello scripting di shell su ogni varietà di sistemi UNIX. La maggior parte dei principi trattati in questo libro si applica altrettanto bene allo scripting con altre shell, quale la Shell Korn, da cui Bash ha derivato alcune delle sue funzionalità [2] e la Shell C e le sue varianti. (si faccia attenzione che programmare con la shell C non è raccomandabile a causa di alcuni problemi ad essa inerenti, come evidenziato da Tom Christiansen in un post su Usenet (http://www.etext.org/Quartz/computer/unix/csh.harmful.gz) nell'Ottobre 1993).

Quello che segue è un manuale sullo scripting di shell che sfrutta i molti esempi per illustrare i diversi elementi della shell. Gli script di esempio funzionano correttamente -- sono stati verificati -- e alcuni di essi possono persino essere impiegati per scopi pratici. Il lettore può divertirsi con il vero codice degli esempi presenti nell'archivio dei sorgenti (`nomescript.sh`), [3] attribuirgli i permessi di esecuzione (con **`chmod u+rx nomescript`**), quindi eseguirli e vedere cosa succede. Se l'archivio dei sorgenti non dovesse essere disponibile, allora si ricorra ad un taglia-incolla dalle versioni HTML, pdf o testo. È da notare che alcuni degli script che seguono introducono degli elementi prima che gli stessi siano stati spiegati e questo richiede, per la loro comprensione, che il lettore dia uno sguardo ai capitoli successivi.

Se non altrimenti specificato, gli script di esempio che seguono sono stati scritti dall'autore del libro.

Notes

1. Ad essi ci si riferisce come builtin, funzionalità interne alla shell.
2. Molti degli elementi di *ksh88* ed anche alcuni della più aggiornata *ksh93* sono stati riuniti in Bash.
3. Convenzionalmente, gli script creati da un utente hanno un nome con estensione `.sh`. Gli script di sistema, come quelli che si trovano nel file `/etc/rc.d`, non seguono questa regola.

Chapter 2. Iniziare con #!

Nel caso più semplice, uno script non è nient'altro che un elenco di comandi di sistema posti in un file. Come minimo si risparmia lo sforzo di ridigitare quella particolare sequenza di comandi tutte le volte che è necessario.

Example 2-1. cleanup: Uno script per cancellare i file di log in /var/log

```
# cleanup
# Da eseguire come root, naturalmente.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Log cancellati."
```

Come si può vedere, qui non c'è niente di insolito, solo una serie di comandi che potrebbero essere eseguiti uno ad uno dalla riga di comando di una console o di un xterm. I vantaggi di collocare dei comandi in uno script vanno, però, ben al di là del non doverli reimmettere ogni volta. Lo script, infatti, può essere modificato, personalizzato o generalizzato per un'applicazione particolare.

Example 2-2. cleanup: Una versione migliorata e generalizzata dello script precedente.

```
#!/bin/bash
# cleanup, versione 2
# Da eseguire come root, naturalmente.

DIR_LOG=/var/log
ROOT_UID=0     # Solo gli utilizzatori con  $UID 0 hanno i privilegi di root.
LINEE=50       # Numero prestabilito di righe salvate.
E_XCD=66       # Riesco a cambiare directory?
E_NONROOT=67   # Codice di exit non-root.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Devi essere root per eseguire questo script."
  exit $E_NONROOT
fi

if [ -n "$1" ]
# Verifica se è presente un'opzione da linea di comando (non-vuota).
then
  linee=$1
else
  linee=$LINEE # Default, se non specificata da riga di comando.
fi


#  Stephane Chazelas suggerisce quello che segue,
#+ come via migliore per la verifica degli argomenti da linea di comando,
#+ ma è ancora un po' prematuro a questo punto del manuale.
```

```
#
#    E_ERR_ARG=65   # Argomento non numerico (formato dell'argomento non valido)
#
#    case "$1" in
#    ""      ) linee=50;;
#    *[!0-9]*) echo "Utilizzo: 'basename $0' file-da-cancellare"; exit\
# $E_ERR_ARG;;
#    *       ) linee=$1;;
#    esac
#
#* Vedere più avanti nel capitolo "Cicli" per la comprensione delle righe
#+ precedenti.


cd $DIR_LOG

if [ 'pwd' != "$DIR_LOG" ]  # o   if [ "$PWD" != "$DIR_LOG" ]
                            # Non siamo in /var/log?
then
  echo "Non riesco a cambiare in $DIR_LOG."
  exit $E_XCD
fi  #  Doppia verifica se si è nella directory corretta, prima di cancellare
    #+ il file di log.

# ancora più efficiente:
#
# cd /var/log || {
#   echo "non riesco a spostarmi nella directory stabilita." >&2
#   exit $E_XCD;
# }



tail -$linee messages > mesg.temp # Salva l'ultima sezione del file di
                                  # log messages.
mv mesg.temp messages             # Diventa la nuova directory di log.


# cat /dev/null > messages
#* Non più necessario, perché il metodo precedente è più sicuro.

cat /dev/null > wtmp  #  ': > wtmp' e '> wtmp'  hanno lo stesso effetto.
echo "Log cancellati."

exit 0
#  Il valore di ritorno zero da uno script
#+ indica alla shell la corretta esecuzione dello stesso.
```

Poiché non si voleva cancellare l'intero log di sistema, questa variante del primo script mantiene inalterata l'ultima sezione del file di log messages. Si scopriranno continuamente altri modi per rifinire questi script ed aumentarne l'efficienza.

I *caratteri* (#!), all'inizio dello script, informano il sistema che il file contiene una serie di comandi che devono essere passati all'interprete indicato. I caratteri #! in realtà sono un "magic number" [1] di due byte, vale a dire un identificatore speciale che designa il tipo di file o, in questo caso, uno script di shell eseguibile (vedi **`man magic`** per ulteriori dettagli su questo affascinante argomento). Immediatamente dopo *#!* si trova il percorso del programma che deve interpretare i comandi contenuti nello script, sia esso una shell, un linguaggio di programmazione o una utility. L'interprete esegue quindi i comandi dello script, iniziando dalla prima riga e ignorando i commenti. [2]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Ognuna delle precedenti intestazioni di script chiama un differente interprete di comandi, sia esso `/bin/sh`, la shell (**bash** in un sistema Linux) o altri. [3] L'utilizzo di **`#!/bin/sh`**, la Bourne Shell predefinita nella maggior parte delle varie distribuzioni commerciali UNIX, rende lo script portabile su macchine non-Linux, sebbene questo significhi sacrificare alcuni elementi specifici di Bash. Lo script sarà, comunque, conforme allo standard POSIX [4] **sh**.

È importante notare che il percorso specificato dopo "#!" deve essere esatto, altrimenti un messaggio d'errore -- solitamente "Command not found" -- sarà l'unico risultato dell'esecuzione dello script.

#! può essere omesso se lo script è formato solamente da una serie di comandi specifici di sistema e non utilizza direttive interne della shell. Il secondo esempio ha richiesto #!, perché la riga di assegnamento di variabile, **`linee=50`**, utilizza un costrutto specifico della shell. È da notare ancora che **`#!/bin/sh`** invoca l'interprete di shell predefinito, che corrisponde a `/bin/bash` su una macchina Linux.

> **Important:** Questo manuale incoraggia l'approccio modulare nella realizzazione di uno script. Si annotino e si raccolgano come "ritagli" i frammenti di codice che potrebbero rivelarsi utili per degli script futuri. Addirittura si potrebbe costruire una libreria piuttosto ampia di routine. Come, ad esempio, la seguente parte introduttiva di uno script che verifica se lo stesso è stato eseguito con il numero corretto di parametri.

```
if [ $# -ne Numero_di_argomenti_attesi ]
then
  echo "Utilizzo: ‘basename $0‘ qualcosa"
  exit $ERR_ARG
fi
```

2.1. Eseguire uno script

Dopo aver creato uno script, lo si può eseguire con **`sh nomescript`** [5] o, in alternativa, con **`bash nomescript`**. Non è raccomandato l'uso di **`sh <nomescript`** perché, così facendo, si disabilita la lettura dallo `stdin`

all'interno dello script. Molto più conveniente è rendere lo script eseguibile direttamente con chmod.

O con:

`chmod 555 nomescript` (che dà a tutti gli utenti il permesso di lettura/esecuzione) [6]

O con

`chmod +rx nomescript` (come il precedente)

`chmod u+rx nomescript` (che attribuisce solo al proprietario dello script il permesso di lettura/esecuzione)

Dopo averlo reso eseguibile, si può verificarne la funzionalità con **`./nomescript`**. [7] Se la prima riga inizia con i caratteri "#!" , all'avvio lo script chiamerà, per la propria esecuzione, l'interprete dei comandi specificato.

Come ultimo passo, dopo la verifica e il debugging, si vorrà probabilmente spostare lo script nella directory `/usr/local/bin` (operazione da eseguire come root) per renderlo disponibile, oltre che per se stessi, anche agli altri utenti, quindi come eseguibile di sistema. In questo modo lo script potrà essere messo in esecuzione semplicemente digitando **nomescript [INVIO]** da riga di comando.

2.2. Esercizi preliminari

1. Gli amministratori di sistema spesso creano degli script per eseguire automaticamente compiti di routine. Si forniscano diversi esempi in cui tali script potrebbero essere utili.
2. Scrivere uno script che all'esecuzione visualizzi l'ora e la data, elenchi tutti gli utenti connessi e fornisca il tempo di esecuzione uptime del sistema. Lo script, quindi, dovrà salvare queste informazioni in un file di log.

Notes

1. Alcune versioni UNIX (quelle basate su 4.2BSD) utilizzano un magic number a quattro byte, che richiede uno spazio dopo il ! -- **`#! /bin/sh`**.
2. La riga con #! dovrà essere la prima cosa che l'interprete dei comandi (**sh** o **bash**) incontra. In caso contrario, dal momento che questa riga inizia con #, verrebbe correttamente interpretata come un commento.

   Se, infatti, lo script include un'*altra riga con* #!, **bash** la interpreterebbe correttamente come un commento, dal momento che il primo #! ha già svolto il suo compito.

   ```
   #!/bin/bash

   echo "Parte 1 dello script."
   a=1

   #!/bin/bash
   # Questo *non* eseguirà un nuovo script.

   echo "Parte 2 dello script."
   ```

```
echo $a  # Il valore di $a è rimasto 1.
```

3. Ciò permette degli ingegnosi espedienti.

```
#!/bin/rm
# Script che si autocancella.

# Niente sembra succedere quando viene eseguito ... solo che il file scompare.
#

QUALUNQUECOSA=65

echo "Questa riga non verrà mai visualizzata (scommettete!)."

exit $QUALUNQUECOSA  # Niente paura. Lo script non terminerà a questo punto.
```

   Provate anche a far iniziare un file `README` con **`#!/bin/more`** e rendetelo eseguibile. Il risultato sarà la visualizzazione automatica del file di documentazione.

4. **P**ortable **O**perating **S**ystem **I**nterface, un tentativo di standardizzare i SO di tipo UNI**X**.
5. Attenzione: richiamando uno script Bash con **`sh nomescript`** si annullano le estensioni specifiche di Bash e, di conseguenza, se ne potrebbe compromettere l'esecuzione.
6. Uno script, per essere eseguito, ha bisogno, oltre che del permesso di esecuzione, anche di quello di *lettura* perché la shell deve essere in grado di leggerlo.
7. Perché non invocare semplicemente uno script con **`nomescript`**? Se la directory in cui ci si trova ($PWD) è anche quella dove *nomescript* è collocato, perché il comando non funziona? Il motivo è che, per ragioni di sicurezza, la directory corrente, ".", non viene inclusa nella variabile $PATH dell'utente. È quindi necessario invocare esplicitamente lo script che si trova nella directory corrente con **`./nomescript`**.

Part 2. I fondamenti

Chapter 3. Caratteri speciali

Caratteri speciali che si trovano negli script e non solo

#

Commenti. Le righe che iniziano con # (con l'eccezione di #!) sono considerate commenti.

```
# Questa riga è un commento.
```

I commenti possono anche essere posti dopo un comando.

```
echo "Seguirà un commento." # Qui il commento.
```

Sono considerati commenti anche quelli che seguono uno o più spazi posti all'inizio di una riga.

```
    # Un carattere di tabulazione precede questo commento.
```

> **Caution**
>
> Un comando non può essere posto dopo un commento sulla stessa riga. Non vi è alcun modo per terminare un commento, in modo che si possa inserire del "codice", da eseguire, sulla stessa riga. È indispensabile inserire il comando in una nuova riga.

Note: Naturalmente, un # preceduto da un carattere di escape in un enunciato **echo** *non* verrà considerato come un commento. Inoltre, un # compare in alcuni costrutti di sostituzione di parametro e nelle espressioni con costanti numeriche.

```
echo "Il presente  # non inizia un commento."
echo 'Il presente  # non inizia un commento.'
echo Il presente  \# non inizia un commento.
echo Il presente   # inizia un commento.

echo ${PATH#*:}       # È una sostituzione di parametro, non un commento.
echo $(( 2#101011 ))  # È una conversione di base, non un commento.

# Grazie, S.C.
```

I caratteri standard per il quoting e l'escaping (" ' \) evitano la reinterpretazione di #.

Anche alcune operazioni di ricerca utilizzano #.

;

Separatore di comandi. [Punto e virgola] Permette di mettere due o più comandi sulla stessa riga.

```
echo ehilà; echo ciao


if [ -x "$nomefile" ]; then    #  Notate che "if" e "then" hanno bisogno del
                               #+ punto e virgola. Perchè?
  echo "Il file $nomefile esiste."; cp $nomefile $nomefile.bak
else
  echo "$nomefile non trovato."; touch $nomefile
fi; echo "Verifica di file completata."
```

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

;;

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

```
case "$variabile" in
abc)  echo "$variabile = abc" ;;
xyz)  echo "$variabile = xyz" ;;
esac
```

.

"punto" comando. Equivale a source (vedi Example 11-19). È un builtin bash.

.

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

```
bash$ touch .hidden-file (file nascosto)
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000
employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
```

```
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

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

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

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

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

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

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

.

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

"

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

’

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

,

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

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

\

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

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

/

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

È anche l'operatore aritmetico di divisione.

‘

sostituzione di comando. [apostrofo inverso] ‘*comando*‘ rende disponibile l'output di *comando* per impostare una variabile. Questo è anche conosciuto come apostrofo inverso o apice inverso.

:

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

```
:
echo $?   # 0
```

Ciclo infinito:

```
while :
do
   operazione-1
   operazione-2
   ...
   operazione-n
done

# Uguale a:
#    while true
#    do
#      ...
#    done
```

Istruzione nulla in un costrutto if/then:

```
if condizione
then :   # Non fa niente e salta alla prossima istruzione
else
```

```
   fa-qualcosa
fi
```

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

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

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

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

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

Espansione di variabile / sostituzione di sottostringa.

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

```
: > data.xxx   # Ora il file "data.xxx" è vuoto.

# Ha lo stesso effetto di cat /dev/null > data.xxx
# Tuttavia non viene generato un nuovo processo poiché ":" è un builtin.
```

Vedi anche Example 12-12.

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

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

Può essere utilizzato per iniziare una riga di commento, sebbene non sia consigliabile. Utilizzando # si disabilita la verifica d'errore sulla parte restante di quella riga, così nulla verrà visualizzato dopo il commento. Questo non è il caso con :.

```
: Questo è un commento che genera un errore, (if [ $x -eq 3] ).
```

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

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

!

inverte (nega) il senso di una verifica o di un exit status. L'operatore ! inverte l'exit status di un comando a cui è stato anteposto (vedi Example 6-2). Cambia anche il significato di un operatore di verifica. Può, per esempio, cambiare il senso di "uguale" (=) in "non uguale" (!=). L'operatore ! è una parola chiave Bash.

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

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

*

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

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

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

*

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

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

?

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

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

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

?

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

$

Sostituzione di variabile.

```
var1=5
var2=23skidoo

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

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

$

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

${}

Sostituzione di parametro.

$*
$@

Parametri posizionali.

$?

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

$$

variabile id di processo. La variabile $$ contiene l’*id di processo* dello script in cui appare.

()

gruppo di comandi.

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

> **Important:** Un elenco di comandi racchiusi da *`parentesi`* dà inizio ad una subshell.
>
> Le variabili all’interno delle parentesi, appartenenti quindi alla subshell, non sono visibili dallo script. Il processo genitore, lo script, non può leggere le variabili create nel processo figlio, la subshell.
>
> ```
> a=123
> ( a=321; )
>
> echo "a = $a"   # a = 123
> # "a" tra parentesi si comporta come una variabile locale.
> ```

inizializzazione di array.

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

{xxx,yyy,zzz,...}

Espansione multipla.

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

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

> **Caution**
>
> Non è consentito alcuno spazio dentro le parentesi, *tranne il caso* in cui si utilizzi il "quoting" o se preceduto da un carattere di escape.
>
> **echo {file1,file2}\ :{\ A," B",' C'}**
>
> file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C

{}

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

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

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (valore di a nel blocco di codice)

# Grazie, S.C.
```

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

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

```
#!/bin/bash
# Legge le righe del file /etc/fstab.

File=/etc/fstab

{
read riga1
read riga2
} < $File

echo "La prima riga di $File è:"
echo "$riga1"
echo
echo "La seconda riga di $File è:"
echo "$riga2"
```

```
exit 0
```

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

```
#!/bin/bash
# rpm-check.sh

# Interroga un file rpm per visualizzarne la descrizione ed il
#+contenuto, verifica anche se può essere installato.
# Salva l’output in un file.
#
# Lo script illustra l’utilizzo del blocco di codice.

SUCCESSO=0
E_ERR_ARG=65

if [ -z "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ file-rpm"
  exit $E_ERR_ARG
fi

{
  echo
  echo "Descrizione Archivio:"
  rpm -qpi $1       # Richiede la descrizione.
  echo
  echo "Contenuto dell’archivio:"
  rpm -qpl $1       # Richiede il contenuto.
  echo
  rpm -i --test $1  # Verifica se il file rpm può essere installato.
  if [ "$?" -eq $SUCCESSO ]
  then
    echo "$1 può essere installato."
  else
    echo "$1 non può essere installato."
  fi
  echo
} > "$1.test"       #  Redirige l’output di tutte le istruzioni del blocco
                    #+ in un file.

echo "I risultati della verifica rpm si trovano nel file $1.test"

# Vedere la pagina di manuale di rpm per la spiegazione delle opzioni.

exit 0
```

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

{} \;

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

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

[ ]

verifica.

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

[[ ]]

verifica.

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

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

[ ]

elemento di un array.

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

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

[ ]

intervallo di caratteri.

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

(( ))

espansione di espressioni intere.

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

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

> &> >& >> <

redirezione.

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

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

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

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

sostituzione di processo.

`(comando)>`

`<(comando)`

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

In un altro ambito ancora, i caratteri "<" e ">" vengono utilizzati come operatori di confronto tra interi. Vedi anche Example 12-7.

<<

redirezione utilizzata in un here document.

<
>

Confronto ASCII.

```
veg1=carote
veg2=pomodori

if [[ "$veg1" < "$veg2" ]]
then
  echo "Sebbene $veg1 preceda $veg2 nel dizionario,"
  echo "questo non intacca le mie preferenze culinarie."
else
  echo "Che razza di dizionario stai usando?"
fi
```

\<
\>

delimitatore di parole in un'espressione regolare.

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

|

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

```
echo ls -l | sh
#  Passa l'output di  "echo ls -l" alla shell,
#+ con lo stesso risultato di "ls -l".


cat *.lst | sort | uniq
# Unisce ed ordina tutti i file ".lst", dopo di che cancella le righe doppie.
```

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

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

```
#!/bin/bash
# uppercase.sh : Cambia l'input in caratteri maiuscoli.

tr 'a-z' 'A-Z'
#  Per l'intervallo delle lettere deve essere utilizzato il "quoting" per
#+ impedire di creare file aventi per nome le singole lettere dei nomi
#+ dei file.

exit 0
```

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

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

> **Note:** In una pipe, lo `stdout` di ogni processo deve essere letto come `stdin` del successivo. Se questo non avviene, il flusso di dati si *blocca*. La pipe non si comporterà come ci si poteva aspettare.
>
> ```
> cat file1 file2 | ls -l | sort
> # L'output proveniente da "cat file1 file2" scompare.
> ```
>
> Una pipe viene eseguita come processo figlio e quindi non può modificare le variabili dello script.
>
> ```
> variabile="valore_iniziale"
> echo "nuovo_valore" | read variabile
> echo "variabile = $variabile"     # variabile = valore_iniziale
> ```
>
> Se uno dei comandi della pipe abortisce, questo ne determina l'interruzione prematura. Chiamata *pipe interrotta*, questa condizione invia un segnale *SIGPIPE*.

>|

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

||

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

&

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

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

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

Example 3-3. Eseguire un ciclo in background

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # Primo ciclo.
do
  echo -n "$i "
done & # Esegue questo ciclo in background.
       # Talvolta verrà eseguito, invece, il secondo ciclo.

echo   # Questo 'echo' alcune volte non verrà eseguito.

for i in 11 12 13 14 15 16 17 18 19 20   # Secondo ciclo.
do
  echo -n "$i "
done

echo   # Questo 'echo' alcune volte non verrà eseguito.

# ======================================================

# Output atteso:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Talvolta si potrebbe ottenere:
# 11 12 13 14 15 16 17 18 19 20
```

```
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (Il secondo 'echo' non è stato eseguito. Perché?)

# Occasionalmente anche:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (Il primo 'echo' non è stato eseguito. Perché?)

#  Molto raramente qualcosa come:
#  11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
#  Il ciclo in primo piano (foreground) ha la precedenza su
#+ quello in background.

exit 0
```

> **Caution**
>
> Un comando eseguito in background all'interno di uno script può provocarne l'interruzione, in attesa che venga premuto un tasto. Fortunatamente, per questa eventualità vi è un rimedio.

&&

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

-

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

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

ls -al

sort -dfu $nomefile

set -- $variabile

if [ $file1 -ot $file2 ]
then
  echo "Il file $file1 è più vecchio di $file2."
fi

if [ "$a" -eq "$b" ]
then
  echo "$a è uguale a $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47 ]
then
  echo "$c è uguale a 24 e $d è uguale a 47."
fi
```

-

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

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Sposta l'intero contenuto di una directory in un'altra
# [cortesia di Alan Cox <a.cox@swansea.ac.uk>, con una piccola modifica]

# 1) cd /source/directory    Directory sorgente, dove sono contenuti i file che
#                            devono essere spostati.
# 2) &&                      "lista And": se l'operazione 'cd' ha successo,
#                            allora viene eseguito il comando successivo.
# 3) tar cf - .              L'opzione 'c' del comando di archiviazione 'tar'
#                            crea un nuovo archivio, l'opzione 'f' (file),
#                            seguita da '-' designa come file di destinazione
#                            lo sdtout, e lo fa nella directory corrente ('.').
# 4) |                       Collegato a...
# 5) ( ... )                 subshell
# 6) cd /dest/directory      Cambia alla directory di destinazione.
# 7) &&                      "lista And", come sopra
# 8) tar xpvf -              Scompatta l'archivio ('x'), mantiene i permessi e
#                            le proprietà dei file ('p'), invia messaggi
#                            dettagliati allo stdout ('v'), leggendo i dati
#                            dallo stdin ('f' seguito da '-')
#                            Attenzione: 'x' è un comando,
#                            mentre 'p', 'v' ed 'f' sono opzioni.
# Whew!



# Più elegante, ma equivalente:
#   cd source-directory
#   tar cf - . | (cd ../dest-directory; tar xzf -)
#
# cp -a /source/directory /dest     anche questo ha lo stesso effetto.


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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

```
#!/bin/bash

#  Salvataggio di tutti i file della directory corrente che sono stati
#+ modificati nelle ultime 24 ore in un archivio "tarball" (file trattato
#+ con tar e gzip).

FILEBACKUP=backup-$(date +%d-%m-%Y)
#                 Inserisce la data nel nome del file di salvataggio.
#                 Grazie a Joshua Tschida per l'idea.
archivio=${1:-$FILEBACKUP}
#  Se non viene specificato un nome di file d'archivio da riga di comando,
#+ questo verrà impostato a "backup-GG-MM-AAAA.tar.gz."

tar cvf - 'find . -mtime -1 -type f -print' > $archivio.tar
gzip $archivio.tar
echo "Directory $PWD salvata nel file \"$archivio.tar.gz\"."


#  Stephane Chazelas evidenzia che il precedente codice fallisce l'esecuzione
#+ se incontra troppi file o se un qualsiasi nome di file contiene caratteri
#+ di spaziatura.
```

```
# Suggerisce, quindi, le seguenti alternative:
# -------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archivio.tar"
#      utilizzando la versione GNU di "find".


#   find . -mtime -1 -type f -exec tar rvf "$archivio.tar" '{}' \;
#         portabile su altre versioni UNIX, ma molto più lento.
# -------------------------------------------------------------------


exit 0
```

> **Caution**
>
> Nomi di file che iniziano con "-" possono provocare problemi quando vengono utilizzati con il "-" come operatore di redirezione. Uno script potrebbe verificare questa possibilità ed aggiungere un prefisso adeguato a tali nomi, per esempio `./-NOMEFILE`, `$PWD/-NOMEFILE` o `$PATHNAME/-NOMEFILE`.
>
> Anche il valore di una variabile che inizia con un `-`, potrebbe creare problemi.
>
> ```
> var="-n"
> echo $var
> # Ha l'effetto di un  "echo -n", che non visualizza nulla.
> ```

-

directory di lavoro precedente. [trattino] **cd -** cambia alla directory di lavoro precedente. Viene utilizzata la variabile d'ambiente $OLDPWD.

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

-

Meno. Segno meno in una operazione aritmetica.

=

Uguale. Operatore di assegnamento

```
a=28
echo $a   # 28
```

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

+

Più. Addizione, operazione aritmetica.

In un contesto differente, il simbolo + è un operatore di espressione regolare.

+

Opzione. Opzione per un comando o un filtro.

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

%

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

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

~

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

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

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

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

~+

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

~-

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

^

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

Caratteri di controllo

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

- **`Ctl-B`**

  Backspace (ritorno non distruttivo).

- **`Ctl-C`**

  Termina un job in primo piano.

- **`Ctl-D`**

  Uscita dalla shell (simile a exit).

  “EOF” (end of file). Termina l’input dallo `stdin`.

- **`Ctl-G`**

  “SEGNALE ACUSTICO” (beep).

- **`Ctl-H`**

  Backspace (ritorno distruttivo).

```
#!/bin/bash
# Inserire Ctl-H in una stringa.

a="^H^H"                  # Due Ctl-H (backspace).
echo "abcdef"             # abcdef
echo -n "abcdef$a "       # abcd f
# Spazio dopo    ^             ^ Doppio backspace
echo -n "abcdef$a"        # abcdef
#  Nessuno spazio alla fine       Non viene seguito il backspace (perché?)
                          # I risultati possono essere piuttosto diversi da
                          #+ ciò che ci si aspetta.
echo; echo
```

- **`Ctl-I`**

  Tabulazione orizzontale.

- **`Ctl-J`**

Nuova riga (line feed).

- **Ctl-K**

  Tabulazione verticale.

- **Ctl-L**

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

- **Ctl-M**

  A capo.

```
#!/bin/bash
# Grazie a Lee Maschmeyer, per l'esempio.

read -n 1 -s -p $'Control-M sposta il cursore all'inizio della riga. Premi Invio. \x0d'
                        # Naturalmente, '0d' è l'equivalente esadecimale di Control-M.
echo >&2   #  '-s' non visualizza quello che viene digitato,
           #+ quindi è necessario andare a capo esplicitamente.

read -n 1 -s -p $'Control-J sposta il cursore alla riga successiva. \x0a'
echo >&2   #  Control-J indica nuova riga (linefeed).

read -n 1 -s -p $'E Control-K\x0b lo sposta direttamente in basso.'
echo >&2   #  Control-K indica la tabulazione verticale.

exit 0
```

- **Ctl-Q**

  Ripristino (XON).

  Ripristina lo `stdin` di un terminale.

- **Ctl-S**

  Sospensione (XOFF).

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

- **Ctl-U**

  Cancella una riga di input.

- **Ctl-Z**

  Sospende un'applicazione in primo piano.

Spaziatura

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

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

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

Notes

1. La shell esegue *l'espansione delle parentesi graffe*. Il comando agisce sul *risultato* dell'espansione.
2. Eccezione: una porzione di codice tra parentesi graffe come parte di una pipe *deve* essere eseguita come subshell.

```
ls | { read primariga; read secondariga; }
#  Errore. Il blocco di codice tra le parentesi graffe esegue una subshell,
#+ così l'output di "ls" non può essere passato alle variabili interne
#  al blocco.

echo "La prima riga è $primariga; la seconda riga è $secondariga"
# Non funziona.

# Grazie, S.C.
```

Chapter 4. Introduzione alle variabili ed ai parametri

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

4.1. Sostituzione di variabile

Il *nome* di una variabile è il contenitore del suo *valore*, il dato memorizzato. Il riferimento a questo valore è chiamato *sostituzione di variabile*.

$

Bisogna fare una netta distinzione tra il *nome* di una variabile ed il suo *valore*. Se **`variabile1`** è il nome di una variabile, allora **`$variable1`** è il riferimento al suo *valore*, il dato in essa contenuto. L'unica volta in cui una variabile compare "nuda", senza il prefisso $, è quando viene dichiarata o al momento dell'assegnamento, quando viene *annullata*, quando viene esportata, o nel caso particolare di una variabile che rappresenta un segnale (vedi Example 30-5). L'assegnamento può essere fatto con l'= (come in *var1=27*), in un enunciato read ed all'inizio di un ciclo (*for var2 in 1 2 3*).

Racchiudere il nome della variabile tra doppi apici (" ") non interferisce con la sostituzione di variabile. Questo viene chiamato quoting parziale, o anche "quoting debole". Al contrario, l'utilizzo degli apici singoli (' ') fa sì che il nome della variabile venga interpretato letteralmente, per cui la sostituzione non avverrà. In questo caso si ha il quoting pieno, chiamato anche "quoting forte". Vedi Chapter 5 per una trattazione dettagliata.

È da notare che **`$variabile`** in realtà è una forma semplificata ed alternativa di **`${variabile}`**. In contesti in cui la sintassi **`$variabile`** può provocare un errore, la forma estesa potrebbe funzionare (vedi Section 9.3, più oltre).

Example 4-1. Assegnamento e sostituzione di variabile

```
#!/bin/bash

# Variabili: assegnamento e sostituzione

a=375
hello=$a

#-----------------------------------------------------------------------
# Quando si inizializzano le variabili, non sono consentiti spazi prima
#+ e dopo il segno  =.

#  Nel caso "VARIABILE =valore",
#+ lo script cerca di eseguire il comando "VARIABILE"  con l'argomento
#+ "=valore". Nel caso "VARIABILE= valore", lo script cerca di eseguire
#+ il comando "valore" con la variabile d'ambiente "VARIABILE" impostata a "".
#-----------------------------------------------------------------------
```

```
echo ciao     # Non è un riferimento a variabile, ma solo la stringa "ciao".

echo $ciao
echo ${ciao}  # Come sopra.

echo "$ciao"
echo "${ciao}"

echo

ciao="A B  C   D"
echo $ciao    # A B C D
echo "$ciao"  # A B  C   D
# Come si può vedere, echo $ciao   e   echo "$ciao" producono
#+ risultati differenti. Il quoting di una variabile conserva gli spazi.

echo

echo ’$ciao’  # $ciao

#  Gli apici singoli disabilitano la referenziazione alla variabile,
#+ perché il simbolo "$" viene interpretato letteralmente.

# Notate l’effetto dei differenti tipi di quoting.


ciao=     # Imposta la variabile al valore nullo.
echo "\$ciao (valore nullo) = $ciao"

#  Attenzione, impostare una variabile al valore nullo non è la stessa
#+ cosa di annullarla, sebbene il risultato finale sia lo stesso (vedi oltre).
#
# ------------------------------------------------------------
#
#  È consentito impostare più variabili sulla stessa riga,
#+ separandole con uno spazio.
#  Attenzione, questa forma può diminuire la leggibilità
#+ e potrebbe non essere portabile.

var1=variabile1  var2=variabile2  var3=variabile3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# Potrebbe causare problemi con le versioni più vecchie di "sh".

# ------------------------------------------------------------

echo; echo

numeri="uno due tre"
altri_numeri="1 2 3"
```

```
# Se ci sono degli spazi all'interno di una variabile, allora è
#+ necessario il quoting.
echo "numeri = $numeri"
echo "altri_numeri = $altri_numeri"   # altri_numeri = 1 2 3
echo

echo "variabile_non_inizializzata = $variabile_non_inizializzata"
# Una variabile non inizializzata ha valore nullo  (nessun valore).
variabile_non_inizializzata=   #  Viene dichiarata, ma non inizializzata
                               #+ (è come impostarla al valore nullo,
                               #+ vedi sopra).
echo "variabile_non_inizializzata = $variabile_non_inizializzata"
                               # Ha ancora valore null.

variabile_non_inizializzata=23       # È impostata.
unset variabile_non_inizializzata    # Viene annullata.
echo "variabile_non_inizializzata = $variabile_non_inizializzata"
                                     # Ha ancora valore nullo.

echo

exit 0
```

Caution

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

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

```
echo "$non_inizializzata"                       # (riga vuota)
let "non_inizializzata += 5"                    # Aggiunge 5 alla variabile.
echo "$non_inizializzata"                       # 5

#  Conclusione:
#  Una variabile non inizializzata non ha alcun valore, tuttavia si comporta,
#+ nelle operazioni aritmetiche, come se il suo valore fosse 0 (zero).
#  Questo è un comportamento non documentato (e probabilmente non portabile).
```

Vedi anche Example 11-20.

4.2. Assegnamento di variabile

=

è l'operatore di assegnamento (*nessuno spazio prima e dopo*)

> **Caution**
>
> Da non confondere con = ed -eq, che servono per le verifiche!
>
> È da notare che l'= può essere sia l'operatore di assegnamento che quello di verifica. Dipende dal contesto in cui si trova.

Example 4-2. Assegnamento esplicito di variabile

```
#!/bin/bash
# Variabili nude

echo

# Quando una variabile è "nuda", cioè, senza il '$' davanti?
# Durante l'assegnamento, ma non nella referenziazione.

# Assegnamento
a=879
echo "Il valore di \"a\" è $a."

# Assegnamento con l'utilizzo di 'let'
let a=16+5
echo "Il valore di \"a\" ora è $a."

echo

# In un ciclo 'for' (in realtà, un tipo di assegnamento mascherato)
echo -n "I valori di \"a\" nel ciclo sono: "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# In un enunciato 'read' (un altro tipo di assegnamento)
echo -n "Immetti il valore di \"a\" "
read a
echo "Il valore di \"a\" ora è $a."

echo

exit 0
```

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

```
#!/bin/bash

a=23              # Caso comune
```

```
echo $a
b=$a
echo $b

# Ora in un modo un po’ più raffinato (sostituzione di comando).

a=‘echo Ciao¡    # Assegna il risultato del comando ’echo’ ad ’a’
echo $a

#  Nota: l’utilizzo del punto esclamativo (!) nella sostituzione di comando
#+ non funziona da riga di comando, perché il (!) attiva il
#+ "meccanismo di cronologia" della shell Bash. All’interno di uno script,
#+ però, le funzioni di cronologia sono disabilitate.

a=‘ls -l‘         #  Assegna il risultato del comando ’ls -l’ ad ’a’
echo $a           #  Senza l’utilizzo del quoting vengono eliminate
                  #+ le tabulazioni ed i ritorni a capo.
echo
echo "$a"         # L’utilizzo del quoting preserva gli spazi.
                  # (Vedi il capitolo sul "Quoting.")

exit 0
```

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

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

4.3. Le variabili Bash non sono tipizzate

A differenza di molti altri linguaggi di programmazione, Bash non differenzia le sue variabili per “tipo”. Essenzialmente le variabili Bash sono stringhe di caratteri, ma, a seconda del contesto, la shell consente le operazioni con interi e i confronti di variabili. Il fattore determinante è se il valore di una variabile è formato solo da cifre.

Example 4-4. Intero o stringa?

```
#!/bin/bash
# int-or-string.sh: Intero o stringa?

a=2334                   # Intero.
let "a += 1"
echo "a = $a "           # a = 2335
echo                     # Intero, ancora.


b=${a/23/BB}             # Sostituisce "23" con "BB".
                         # Questo trasforma $b in una stringa.
```

```
echo "b = $b"           # b = BB35
declare -i b            # Dichiararla come intero non aiuta.
echo "b = $b"           # b = BB35

let "b += 1"            # BB35 + 1 =
echo "b = $b"           # b = 1
echo

c=BB34
echo "c = $c"           # c = BB34
d=${c/BB/23}            # Sostituisce "BB" con "23".
                        # Questo trasforma $d in un intero.
echo "d = $d"           # d = 2334
let "d += 1"            # 2334 + 1 =
echo "d = $d"           # d = 2335
echo

# Che dire sulle variabili nulle?
e=""
echo "e = $e"           #  e =
let "e += 1"            #  Sono consentite le operazioni aritmetiche sulle
                        #+ variabili nulle?
echo "e = $e"           #  e = 1
echo                    #  Variabile nulla trasformata in un intero.

# E sulle variabili non dichiarate?
echo "f = $f"           # f =
let "f += 1"            # Sono consentite le operazioni aritmetiche?
echo "f = $f"           # f = 1
echo                    # Variabile non dichiarata trasformata in un intero.



# Le variabili in Bash non sono tipizzate.

exit 0
```

Le variabili non tipizzate sono sia una benedizione che una calamità. Permettono maggiore flessibilità nello scripting (abbastanza corda per impiccarvici!) e rendono più semplice sfornare righe di codice. Per contro, consentono errori subdoli e incoraggiano stili di programmazione disordinati.

È compito del programmatore tenere traccia dei tipi di variabili contenute nello script. Bash non lo farà per lui.

4.4. Tipi speciali di variabili

`variabili locali`

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

variabili d'ambiente

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

> **Note:** Più in generale, ogni processo ha un "ambiente", ovvero un gruppo di variabili contenenti informazioni a cui il processo fa riferimento. Da questo punto di vista, la shell si comporta come qualsiasi altro processo.
>
> Tutte le volte che la shell viene eseguita, crea le variabili di shell che corrispondono alle sue variabili d'ambiente. L'aggiornamento o l'aggiunta di nuove variabili di shell provoca l'aggiornamento del suo ambiente. Tutti i processi generati dalla shell (i comandi eseguiti) ereditano questo ambiente.

> **Caution**
>
> Lo spazio assegnato all'ambiente è limitato. Creare troppe variabili d'ambiente, o se alcune occupano eccessivo spazio, potrebbe causare problemi.
>
> ```
> bash$ eval "'seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZZZZZ/''"
>
> bash$ du
> bash: /usr/bin/du: Argument list too long
> ```
>
> (Grazie, S. C. per i chiarimenti e per aver fornito l'esempio.)

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

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

---

parametri posizionali

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

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

Example 4-5. Parametri posizionali

```
#!/bin/bash

# Eseguite lo script con almeno 10 parametri, per esempio
# ./nomescript 1 2 3 4 5 6 7 8 9 10
```

```
MINPARAM=10

echo

echo "Il nome dello script è \"$0\"."
# Aggiungete ./ per indicare la directory corrente
echo "Il nome dello script è \"‘basename $0‘\"."
# Visualizza il percorso del nome (vedi ’basename’)

echo

if [ -n "$1" ]               #  Utilizzate il quoting per la variabile
                             #+ da verificare.
then
 echo "Il parametro #1 è $1"  # È necessario il quoting
                             #+ per visualizzare il #
fi

if [ -n "$2" ]
then
 echo "Il parametro #2 è $2"
fi

if [ -n "$3" ]
then
 echo "Il parametro #3 è $3"
fi

# ...


if [ -n "${10}" ]  #  I parametri > $9 devono essere racchiusi
                   #+ tra {parentesi graffe}.
then
 echo "Il parametro #10 è ${10}"
fi

echo "-----------------------------------"
echo "In totale i parametri passati sono: "$*""

if [ $# -lt "$MINPARAM" ]
then
  echo
  echo "Occorre fornire almeno $MINPARAM argomenti da riga di comando!"
fi

echo

exit 0
```

La *notazione parentesi graffe*, applicata ai parametri posizionali, è una forma abbastanza semplice di referenziazione all’*ultimo* argomento passato allo script da riga di comando. Questo richiede anche la referenziazione indiretta.

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

Alcuni script possono eseguire compiti diversi, a seconda del nome con cui vengono invocati. Affinché questo possa avvenire, lo script ha necessità di verificare `$0`, cioè il nome con cui è stato invocato. Naturalmente devono esserci dei link simbolici ai nomi alternativi dello script.

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

```
variabile1_=$1_
#  Questo evita qualsiasi errore, anche se non è presente
#+ il parametro posizionale.

argomento_critico01=$variabile1_

#  Il carattere aggiunto può essere tolto più tardi, se si
#+ desidera, in questo modo:
variabile1=${variabile1_/_/}   #  Si hanno effetti collaterali solo se
                               #+ $variabile1_ inizia con "_".
#  È stato utilizzato uno dei modelli di sostituzione di parametro trattati
#+ nel Capitolo 9. Un modo più diretto per gestire la situazione è una
#+ semplice verifica della presenza dei parametri posizionali attesi.

if [ -z $1 ]
then
  exit $MANCA_PARAM_POS
fi

---
```

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

```
#!/bin/bash

# Esegue una verifica ’whois nome-dominio’ su uno dei 3 server:
#                    ripe.net, cw.net, radb.net

# Mettete questo script, con nome ’wh’ nel file /usr/local/bin

# Sono richiesti i seguenti link simbolici:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb


if [ -z "$1" ]
```

```
then
  echo "Utilizzo: ‘basename $0‘ [nome-dominio]"
  exit 65
fi

case ‘basename $0‘ in
# Verifica il nome dello script e interroga il server adeguato
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "Utilizzo: ‘basename $0‘ [nome-dominio]";;
esac

exit 0

---
```

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

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

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

Example 4-7. Utilizzare shift

```
#!/bin/bash
# Utilizzo di ’shift’ per elaborare tutti i parametri posizionali.

#  Chiamate lo script shft ed invocatelo con alcuni parametri, per esempio
#          ./shft a b c def 23 skidoo

until [ -z "$1" ]  # Finché ci sono parametri...
do
  echo -n "$1 "
  shift
done

echo               # Linea extra.

exit 0
```

Note: Il comando **shift** funziona anche sui parametri passati ad una funzione. Vedi Example 34-11.

Notes

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

Chapter 5. Quoting

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

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

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

> **Note:** Tuttavia alcuni programmi ed utility possono ancora reinterpretare o espandere i caratteri speciali contenuti in una stringa a cui è stato applicato il quoting. Questo rappresenta un importante utilizzo del quoting: proteggere un parametro passato da riga di comando dalla reinterpretazione da parte della shell, ma permettere ancora al programma chiamante di espanderlo.
>
> ```
> bash$ grep '[Pp]rima' *.txt
> file1.txt:Questa è la prima riga di file1.txt.
> file2.txt:Questa è la prima riga di file2.txt.
> ```
>
> È da notare che l'istruzione **`grep [Pp]rima *.txt`**, senza quoting, funziona con la shell Bash, ma *non* con **tcsh**.

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

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

```
variabile1="una variabile contenente cinque parole"
COMANDO Questa è $variabile1    # Esegue COMANDO con 7 argomenti:
# "Questa" "è" "una" "variabile" "contenente" "cinque" "parole"

COMANDO "Questa è $variabile1"  # Esegue COMANDO con 1 argomento:
# "Questa è una variabile contenente cinque parole"


variabile2=""    # Vuota.

COMANDO $variabile2 $variabile2 $variabile2
```

```
# Esegue COMANDO con nessun argomento.

COMANDO "$variabile2" "$variabile2" "$variabile2"
# Esegue COMANDO con 3 argomenti vuoti.

COMANDO "$variabile2 $variabile2 $variabile2"
# Esegue COMANDO con 1 argomento (i 2 spazi).

# Grazie, S.C.
```

> **Tip:** È necessario porre gli argomenti tra doppi apici in un enunciato **echo** solo quando la suddivisione delle parole può rappresentare un problema.

Example 5-1. Visualizzare variabili strane

```
#!/bin/bash
# weirdvars.sh: Visualizzare strane variabili.

var="’(]\\{}\$\""
echo $var        # ’(]\{}$"
echo "$var"      # ’(]\{}$"     Nessuna differenza.

echo

IFS=’\’
echo $var        # ’(] {}$"     \ trasformata in spazio.
echo "$var"      # ’(]\{}$"

# Esempi forniti da S.C.

exit 0
```

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

> **Note:** Dal momento che anche il carattere di escape (\) viene interpretato letteralmente, effettuare il quoting di apici singoli mediante apici singoli non produce il risultato atteso.
>
> ```
> echo "Why can’t I write ’s between single quotes"
> # Perché non riesco a scrivere ’s tra apici singoli
>
> echo
>
> # Metodo indiretto.
> echo ’Why can’\”t I write ’"’"’s between single quotes’
> #    |-------|  |----------|   |-----------------------|
> #  Tre stringhe tra apici singoli, con il carattere di escape e
> ```

```
#+ apice singolo in mezzo.
#  Esempio cortesemente fornito da Stephane Chazelas.
```

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

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

Significati speciali di alcuni caratteri preceduti da quello di escape:

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

\n

significa a capo

\r

significa invio

\t

significa tabulazione

\v

significa tabulazione verticale

\b

significa ritorno (backspace)

\a

"significa allerta" (beep o accensione di un led)

\0xx

trasforma in carattere ASCII il valore ottale *0xx*

Example 5-2. Sequenze di escape

```
#!/bin/bash
# escaped.sh: sequenze di escape

echo; echo
```

```
echo "\v\v\v\v\v"      # Visualizza letteralmente: \v\v\v\v\v .
#  Utilizzate l'opzione  -e  con 'echo' per un corretto utilizzo delle
#+ sequenze di escape.
echo "=============="
echo "TABULAZIONE VERTICALE"
echo -e "\v\v\v\v\v"   # Esegue 4 tabulazioni verticali.
echo "==============="

echo "VIRGOLETTE"
echo -e "\042"       #  Visualizza " (42 è il valore ottale del
                     #+ carattere ASCII virgolette).
echo "==============="

# Il costrutto $'\X'  rende l'opzione -e superflua.
echo; echo "A_CAPO E BEEP"
echo $'\n'           # A capo.
echo $'\a'           # Allerta (beep).

echo "================"
echo "VIRGOLETTE"
# La versione 2 e successive di Bash consente l'utilizzo del costrutto $'\nnn'.
# Notate che in questo caso, '\nnn' è un valore ottale.
echo $'\t \042 \t'   # Virgolette (") tra due tabulazioni.

# Può essere utilizzato anche con valori esadecimali nella forma $'\xhhh'.
echo $'\t \x22 \t'   # Virgolette (") tra due tabulazioni.
# Grazie a Greg Keraunen per la precisazione.
# Versioni precedenti di Bash consentivano '\x022'.
echo "================"
echo


# Assegnare caratteri ASCII ad una variabile.
# -------------------------------------------
virgolette=$'\042'   # " assegnate alla variabile.
echo "$virgolette Questa è una stringa tra virgolette $virgolette, \
mentre questa parte è al di fuori delle virgolette."

echo

# Concatenare caratteri ASCII in una variabile.

tripla_sottolineatura=$'\137\137\137'
# 137 è il valore ottale del carattere ASCII  '_'.
echo "$tripla_sottolineatura SOTTOLINEA  $tripla_sottolineatura"

echo

ABC=$'\101\102\103\010'
# 101, 102, 103 sono i valori ottali di A, B, C.

echo $ABC
```

```
echo; echo

escape=$'\033'
# 033 è il valore ottale del carattere di escape.

echo "\"escape\" visualizza come $escape"
# nessun output visibile.

echo; echo

exit 0
```

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

\"

mantiene il significato letterale dei doppi apici

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

\$

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

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

\\

mantiene il significato letterale della barra inversa

```
echo "\\"  # visualizza \

# Mentre . . .

echo "\"   # Invoca il prompt secondario, da riga di comando.
           # In uno script, dà un messaggio d'errore.
```

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

```
                     #  Escaping e quoting semplice
echo \z              #  z
echo \\z             # \z
```

```
echo ’\z’             # \z
echo ’\\z’            # \\z
echo "\z"             # \z
echo "\\z"            # \z

                      #  Sostituzione di comando
echo ‘echo \z‘        #  z
echo ‘echo \\z‘       #  z
echo ‘echo \\\z‘      # \z
echo ‘echo \\\\z‘     # \z
echo ‘echo \\\\\\z‘   # \z
echo ‘echo \\\\\\\z‘  # \\z
echo ‘echo "\z"‘      # \z
echo ‘echo "\\z"‘     # \z

                      # Here document
cat <<EOF
\z
EOF                   # \z

cat <<EOF
\\z
EOF                   # \z

# Esempi forniti da Stephane Chazelas.
```

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

```
variabile=\
echo "$variabile"
# Non funziona - dà un messaggio d’errore:
# test.sh: : command not found
# Un escape "nudo" non può essere assegnato in modo sicuro ad una variabile.
#
#  Quello che avviene effettivamente qui è che la "\" esegue l’escape
#+ del a capo e l’effetto è         variabile=echo "$variabile"
#+                                  assegnamento di variabile non valido

variabile=\
23skidoo
echo "$variabile"       #  23skidoo
                        #  Funziona, perché nella seconda riga
                        #+ è presente un assegnamento di variabile valido.

variabile=\
#        \^    escape seguito da uno spazio
echo "$variabile"       # spazio

variabile=\\
echo "$variabile"       # \

variabile=\\\
echo "$variabile"
# Non funziona - dà un messaggio d’errore:
# test.sh: \: command not found
#
```

```
#  Il primo carattere di escape esegue l’escaping del secondo, mentre il terzo
#+ viene lasciato "nudo", con l’identico risultato del primo esempio visto
#+ sopra.

variabile=\\\\
echo "$variabile"       #  \\
                        #  Il secondo ed il quarto sono stati preservati dal
                        #+ primo e dal terzo.
                        #  Questo è o.k.
```

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

```
elenco_file="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# Elenco di file come argomento(i) di un comando.

# Aggiunge due file all’elenco, quindi visualizza tutto.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $elenco_file

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

# Cosa succede se si effettua l’escaping dei due spazi?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $elenco_file
#  Errore: i primi tre file vengono concatenati e considerati come un unico
#+ argomento per ’ls -l’ perché l’escaping dei due spazi impedisce la
#+ divisione degli argomenti (parole).
```

Il carattere di escape rappresenta anche un mezzo per scrivere comandi su più righe. Di solito, ogni riga rappresenta un comando differente, ma il carattere di escape posto in fine di riga *effettua l’escaping del carattere a capo*, in questo modo la sequenza dei comandi continua alla riga successiva.

```
((cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Ripetizione del comando copia di un albero di directory di Alan Cox,
# ma suddiviso su due righe per aumentarne la leggibilità.

# Come alternativa:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# Vedi la nota più sotto.
#(Grazie, Stephane Chazelas.)
```

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

```
echo "foo
bar"
#foo
#bar

echo

echo ’foo
bar’    # Ancora nessuna differenza.
#foo
#bar

echo

echo foo\
bar     # Eseguito l’escaping del carattere a capo.
#foobar

echo

echo "foo\
bar"     #  Stesso risultato, perché \ viene ancora interpretato come escape
         #+ quando è posto tra apici doppi.
#foobar

echo

echo ’foo\
bar’     #  Il carattere di escape \ viene interpretato letteralmente a causa
         #+ del quoting forte.
#foo\
#bar

# Esempi suggeriti da Stephane Chazelas.
```

Notes

1. Racchiudere il “!” tra doppi apici provoca un errore se usato *da riga di comando*. Apparentemente viene interpretato come un comando di cronologia. In uno script, tuttavia, questo problema non si presenta.

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

   ```
   bash$ echo ciao\!
   ciao!


   bash$ echo "ciao\!"
   ciao\!
   ```

```
bash$ echo -e x\ty
xty


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

(Grazie a Wayne Pollock per la precisazione.)

2. La “divisione delle parole”, in questo contesto, significa suddividere una stringa di caratteri in un certo numero di argomenti separati e distinti.

Chapter 6. Exit ed exit status

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

Chet Ramey

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

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

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

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

```
#!/bin/bash

COMANDO_1

. . .

# Esce con lo status dell'ultimo comando.
ULTIMO_COMANDO

exit
```

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

```
#!/bin/bash

COMANDO_1

. . .

# Esce con lo status dell'ultimo comando.
ULTIMO_COMANDO

exit $?
```

```
#!/bin/bash

COMANDO1
```

```
. . .

# Esce con lo status dell'ultimo comando.
ULTIMO_COMANDO
```

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

Example 6-1. exit / exit status

```
#!/bin/bash

echo ciao
echo $?    # Exit status 0 perché il comando è stato eseguito con successo.

lskdf      #  Comando sconosciuto.
echo $?    #  Exit status diverso da zero perché il comando non ha
           #+ avuto successo.

echo

exit 113   # Restituirà 113 alla shell.
           # Per verificarlo, digitate "echo $?" dopo l'esecuzione dello script.

#  Convenzionalmente, un 'exit 0' indica successo,
#+ mentre un valore diverso significa errore o condizione anomala.
```

$? è particolarmente utile per la verifica del risultato di un comando in uno script (vedi Example 12-28 e Example 12-14).

> **Note:** Il !, l'operatore logico "not", inverte il risultato di una verifica o di un comando e questo si riperquote sul relativo exit status.
>
> **Example 6-2. Negare una condizione utilizzando !**
>
> ```
> true  # il builtin "true".
> echo "exit status di \"true\" = $?"     # 0
>
> ! true
> echo "exit status di \"! true\" = $?"   # 1
> # Notate che "!" deve essere seguito da uno spazio.
> # !true restituisce l'errore  "command not found"
>
> # Grazie, S.C.
> ```

Caution

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

Chapter 7. Verifiche

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

7.1. Costrutti condizionali

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

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

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

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

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

  ```
  if cmp a b &> /dev/null  # Sopprime l'output.
  then echo "I file a e b sono identici."
  else echo "I file a e b sono diversi."
  fi

  if grep -q Bash file
  then echo "Il file contiene almeno un'occorrenza di Bash."
  fi

  if COMANDO_CON_EXIT_STATUS_0_SE_NON_SI_VERIFICA_UN_ERRORE
  then echo "Comando eseguito."
  else echo "Comando fallito."
  fi
  ```

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

  ```
  if echo "Il prossimo *if* è parte del costrutto del primo *if*."

    if [[ $confronto = "intero" ]]
      then (( a < b ))
  ```

```
    else
      [[ $a < $b ]]
    fi

  then
    echo ’$a è inferiore a $b’
  fi
```

Dettagliata spiegazione della “condizione-if” cortesia di Stephane Chazelas.

Example 7-1. Cos’è vero?

```
#!/bin/bash

echo

echo "Verifica \"0\""
if [ 0 ]      # zero
then
  echo "0 è vero."
else
  echo "0 è falso."
fi            # 0 è vero.

echo

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

echo

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

echo

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

```
  echo "NULL è falso."
fi            # NULL è falso.

echo

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

echo

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

echo

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

echo


xyz=          # Inizializzata, ma impostata a valore nullo.

echo "Verifica \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "La variabile nulla è vero."
else
  echo "La variabile nulla è falso."
fi            # La variabile nulla è falso.


echo


# Quando "falso" è vero?
```

```
echo "Verifica \"falso\""
if [ "falso" ]              #  Sembra che "falso" sia solo una stringa.
then
  echo "\"falso\" è vero."  # e verifica se è vero.
else
  echo "\"falso\" è falso."
fi            # "falso" è vero.

echo

echo "Verifica \"\$falso\""  # Ancora variabile non inizializzata.
if [ "$falso" ]
then
  echo "\"\$falso\" è vero."
else
  echo "\"\$falso\" è falso."
fi            # "$falso" è falso.
              # Ora abbiamo ottenuto il risultato atteso.


echo

exit 0
```

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

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

> **Note:** Quando *if* e *then* sono sulla stessa riga, occorre mettere un punto e virgola dopo l'enunciato *if* per indicarne il termine. Sia *if* che *then* sono parole chiave. Le parole chiave (o i comandi) iniziano gli enunciati e prima che un nuovo enunciato possa incominciare sulla stessa riga, è necessario che il precedente venga terminato.
>
> ```
> if [ -x "$nome_file" ]; then
> ```

Else if ed elif

elif

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

```
if [ condizione1 ]
then
   comando1
   comando2
   comando3
elif [ condizione2 ]
# Uguale a else if
then
   comando4
   comando5
else
   comando-predefinito
fi
```

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

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

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

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

```
#!/bin/bash

echo

if test -z "$1"
```

```
then
  echo "Nessun argomento da riga di comando."
else
  echo "Il primo argomento da riga di comando è $1."
fi

echo

if /usr/bin/test -z "$1"      # Stesso risultato del builtin "test".
then
  echo "Nessun argomento da riga di comando."
else
  echo "Il primo argomento da riga di comando è $1."
fi

echo

if [ -z "$1" ]                #  Funzionalità identica al precedente blocco
                              #+ di codice.
#   if [ -z "$1"                 dovrebbe funzionare, ma...
#+  Bash risponde con il messaggio d'errore di missing close-bracket.
then
  echo "Nessun argomento da riga di comando."
else
  echo "Il primo argomento da riga di comando è $1."
fi

echo

if /usr/bin/[ -z "$1"         # Ancora, funzionalità identica alla precedente.
# if /usr/bin/[ -z "$1" ]     # Funziona, ma dà un messaggio d'errore.
then
  echo "Nessun argomento da riga di comando."
else
  echo "Il primo argomento da riga di comando è $1."
fi

echo

exit 0
```

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

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

```
file=/etc/passwd

if [[ -e $file ]]
then
```

```
  echo "Il file password esiste."
fi
```

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

> **Note:** Dopo un **if** non sono strettamente necessari né il comando **test** né i costrutti parentesi quadre ([ ] o [[ ]]).
>
> ```
> dir=/home/bozo
>
> if cd "$dir" 2>/dev/null; then # "2>/dev/null" sopprime il messaggio d'errore.
>   echo "Ora sei in $dir."
> else
>   echo "Non riesco a cambiare in $dir."
> fi
> ```
>
> Il costrutto "if COMANDO" restituisce l'exit status di COMANDO.
>
> Per questo motivo, una condizione tra parentesi quadre può essere utilizzata da sola, senza **if**, se abbinata ad un costrutto lista.
>
> ```
> var1=20
> var2=22
> [ "$var1" -ne "$var2" ] && echo "$var1 è diversa da $var2"
>
> home=/home/bozo
> [ -d "$home" ] || echo "La directory $home non esiste."
> ```

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

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

```
#!/bin/bash
# Verifiche aritmetiche.

# Il costrutto (( ... )) valuta e verifica le espressioni aritmetiche.
# Exit status opposto a quello fornito dal costrutto  [ ... ]!

(( 0 ))
echo "l'exit status di \"(( 0 ))\" è $?."     # 1

(( 1 ))
echo "L'exit status di \"(( 1 ))\" è $?."     # 0
```

```
(( 5 > 4 ))                                    # vero
echo "L'exit status di \"(( 5 > 4 ))\" è $?." # 0

(( 5 > 9 ))                                    # falso
echo "L'exit status di \"(( 5 > 9 ))\" è $?." # 1

(( 5 - 5 ))                                    # 0
echo "L'exit status di \"(( 5 - 5 ))\" è $?." # 1

(( 5 / 4 ))                                    # Divisione o.k.
echo "L'exit status di \"(( 5 / 4 ))\" è $?." # 0

(( 1 / 2 ))                                    # Risultato della divisione <1.
echo "L'exit status di \"(( 1 / 2 ))\" è $?." # Arrotondato a 0.
                                               # 1

(( 1 / 0 )) 2>/dev/null                        # Divisione per 0 non consentita.
echo "L'exit status di \"(( 1 / 0 ))\" è $?." # 1

# Che funzione ha "2>/dev/null"?
# Cosa succederebbe se fosse tolto?
# Toglietelo, quindi rieseguite lo script.

exit 0
```

7.2. Operatori di verifica di file

Restituiscono vero se...

-e

il file esiste

-f

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

-s

il file ha dimensione superiore a zero

-d

il file è una directory

-b

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

-c

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

-p

il file è una pipe

-h

il file è un link simbolico

-L

il file è un link simbolico

-S

il file è un socket

-t

il file (descrittore) è associato ad un terminale

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

-r

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

-w

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

-x

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

-g

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

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

-u

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

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

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

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

-k

è impostato lo *sticky bit*

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

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

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

-O

siete il proprietario del file

-G

l’id di gruppo del file è uguale al vostro

-N

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

f1 -nt f2

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

f1 -ot f2

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

f1 -ef f2

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

!

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

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

```
#!/bin/bash
# broken-link.sh
# Scritto da Lee Bigelow <ligelowbee@yahoo.com>
# Utilizzato con il consenso dell’autore.

#  Uno script di pura shell per cercare i link simbolici "morti" e visualizzarli
#+ tra virgolette, in modo tale che possano essere trattati e dati in pasto a
#+ xargs  :)    es. broken-link.sh /unadirectory  /altradirectory | xargs rm
```

```
#
#Il seguente, tuttavia, è il metodo migliore:
#
#find "unadirectory" -type l -print0|\
#xargs -r0 file|\
#grep "link simbolici interrotti"|
#sed -e 's/^\|: *link simbolici interrotti.*$/"/g'
#
#ma non sarebbe bash pura, come deve essere.
#Prudenza: state attenti al file system /proc e a tutti i link circolari!
################################################################################


#  Se nessun argomento viene passato allo script, la directory di ricerca
#+ directorys viene impostata alla directory corrente. Altrimenti directorys
#+ viene impostata all'argomento passato.
########################################
[ $# -eq 0 ] && directorys=`pwd` || directorys=$@

#  Implementazione della funzione linkchk per cercare, nella directory
#  passatale, i file che sono link a file inesistenti, quindi visualizzarli
#+ tra virgolette. Se uno degli elementi della directory è una sottodirectory,
#+ allora anche questa viene passata alla funzione linkchk.
##########
linkchk () {
    for elemento in $1/*; do
    [ -h "$elemento" -a ! -e "$elemento" ] && echo \"$elemento\"
    [ -d "$elemento" ] && linkchk $elemento
    # Naturalmente, '-h' verifica i link simbolici, '-d' le directory.
    done
}

#  Invia ogni argomento passato allo script alla funzione linkchk, se è una
#+ directory valida. Altrimenti viene visualizzato un messaggio d'errore e le
#+ informazioni sull'utilizzo.
############################
for directory in $directorys; do
    if [ -d $directory ]
then linkchk $directory
else
    echo "$directory non è una directory"
    echo "Utilizzo: $0 dir1 dir2 ..."
    fi
done

exit 0
```

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

7.3. Operatori di confronto (binari)

confronto di interi

-eq

è uguale a

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

-ne

è diverso (non uguale) da

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

-gt

è maggiore di

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

-ge

è maggiore di o uguale a

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

-lt

è minore di

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

-le

è minore di o uguale a

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

<

è minore di (tra doppie parentesi)

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

<=

è minore di o uguale a (tra doppie parentesi)

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

>

è maggiore di (tra doppie parentesi)

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

>=

è maggiore di o uguale a (tra doppie parentesi)

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

confronto di stringhe

=

è uguale a

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

==

è uguale a

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

È sinonimo di =.

```
[[ $a == z* ]]    # vero se $a inizia con una "z" (ricerca di corrispondenza)
[[ $a == "z*" ]]  # vero se $a è uguale a z*

[ $a == z* ]      # esegue il globbing e la divisione delle parole
[ "$a" == "z*" ]  # vero se $a è uguale a z*

# Grazie, S.C.
```

!=

è diverso (non uguale) da

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

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

<

è inferiore a, in ordine alfabetico ASCII

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

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

>

è maggiore di, in ordine alfabetico ASCII

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

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

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

-z

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

-n

la stringa non è "nulla".

> **Caution**
>
> L'operatore **-n** richiede assolutamente il quoting della stringa all'interno delle parentesi quadre. L'utilizzo, tra le parentesi quadre, di una stringa senza quoting sia con **! -z**, che da sola (vedi Example 7-6) normalmente funziona, tuttavia non è una pratica sicura. Bisogna *sempre* utilizzare il quoting su una stringa da verificare. [3]

Example 7-5. Confronti aritmetici e di stringhe

```
#!/bin/bash

a=4
b=5

#  Qui "a" e "b" possono essere trattate sia come interi che come stringhe.
#  Ci si può facilmente confondere tra i confronti aritmetici e quelli sulle
#+ stringhe, perché le variabili Bash non sono tipizzate.
#
#  Bash consente le operazioni di interi e il confronto di variabili
```

```
#+ il cui valore è composto solamente da cifre.
#  Attenzione, siete avvisati.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a non è uguale a $b"
  echo "(confronto aritmetico)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a non è uguale a $b."
  echo "(confronto di stringhe)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# In questo particolare esempio funziona sia "-ne" che "!=".

echo

exit 0
```

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

```
#!/bin/bash
# str-test.sh: Verifica di stringhe nulle e di stringhe senza quoting (*)

# Utilizzando   if [ ... ]


# Se una stringa non è stata inizializzata, non ha un valore definito.
# Questo stato si dice "nullo" (non zero!).

if [ -n $stringa1 ]    # $stringa1 non è stata dichiarata o inizializzata.
then
  echo "Stringa \"stringa1\" non è nulla."
else
  echo "Stringa \"stringa1\" è nulla."
fi

# Risultato sbagliato.
# Viene visualizzato $stringa1 come non nulla, anche se non era inizializzata.

echo


# Proviamo ancora.
```

```
if [ -n "$stringa1" ]  # Questa volta è stato applicato il quoting a $stringa1.
then
  echo "Stringa \"stringa1\" non è nulla."
else
  echo "Stringa \"stringa1\" è nulla."
fi                     #  Usate il quoting per le stringhe nel costrutto
                       #+ di verifica parentesi quadre!


echo


if [ $stringa1 ]       # Qui, $stringa1 è sola.
then
  echo "Stringa \"stringa1\" non è nulla."
else
  echo "Stringa \"stringa1\" è nulla."
fi

#  Questo funziona bene.
#  L'operatore di verifica [ ] da solo è in grado di rilevare se la stringa
#+ è nulla.
#  Tuttavia è buona pratica usare il quoting ("$stringa1").
#
# Come ha evidenziato Stephane Chazelas,
#    if [ $stringa1 ]    ha un argomento, "]"
#    if [ "$stringa1" ]  ha due argomenti, la stringa vuota "$stringa1" e "]"


echo



stringa1=inizializzata

if [ $stringa1 ]       # Ancora, $stringa1 da sola.
then
  echo "Stringa \"stringa1\" non è nulla."
else
  echo "Stringa \"stringa1\" è nulla."
fi

# Ancora, risultato corretto.
# Nondimeno, è meglio utilizzare il quoting ("$stringa1"), perché. . .

stringa1="a = b"

if [ $stringa1 ]       # Ancora $stringa1 da sola.
then
  echo "Stringa \"stringa1\" non è nulla."
else
  echo "Stringa \"stringa1\" è nulla."
```

```
fi

# Senza il quoting di "$stringa1" ora si ottiene un risultato sbagliato!

exit 0

# Grazie anche a Florian Wisser per la "citazione iniziale".

# (*) L’intestazione di commento originaria recita "Testing null strings
#+ and unquoted strings, but not strings and sealing wax, not to
# mention cabbages and kings ..." attribuita a Florian Wisser. La
# seconda riga non è stata tradotta in quanto, la sua traduzione
# letterale, non avrebbe avuto alcun senso nel contesto attuale
# (N.d.T.).
```

Example 7-7. zmost

```
#!/bin/bash

# Visualizza i file gzip con  ’most’

NOARG=65
NONTROVATO=66
NONGZIP=67

if [ $# -eq 0 ] # stesso risultato di:  if [ -z "$1" ]
# $1 può esserci, ma essere vuota:  zmost "" arg2 arg3
then
  echo "Utilizzo: ‘basename $0‘ nomefile" >&2
  # Messaggio d’errore allo stderr.
  exit $NOARG
  # Restituisce 65 come exit status dello script (codice d’errore).
fi

nomefile=$1

if [ ! -f "$nomefile" ]   # Il quoting di $nomefile mantiene gli spazi.
then
  echo "File $nomefile non trovato!" >&2
  # Messaggio d’errore allo stderr.
  exit $NONTROVATO
fi

if [ ${nomefile##*.} != "gz" ]
# Uso delle parentesi graffe nella sostituzione di variabile.
then
  echo "Il file $1 non è un file gzip!"
  exit $NONGZIP
fi

zcat $1 | most
```

```
# Usa il visualizzatore di file  'most' (simile a 'less').
# Le versioni più recenti di 'most' hanno la capacità di decomprimere un file.
# Si può sostituire con 'more' o 'less', se si desidera.


exit $?   # Lo script restituisce l'exit status della pipe.
# In questo punto dello script  "exit $?" è inutile perché lo script,
# in ogni caso, restituirà l'exit status dell'ultimo comando eseguito.
```

confronti composti

-a

and logico

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

-o

or logico

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

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

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

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

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

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

7.4. Costrutti condizionali if/then annidati

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

```
if [ condizione1 ]
then
  if [ condizione2 ]
  then
    fa-qualcosa  # Ma solo se sia "condizione1" che "condizione2" sono vere.
  fi
fi
```

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

7.5. Test sulla conoscenza delle verifiche

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

```
if [ -f $HOME/.Xclients ]; then
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
     # failsafe settings.  Although we should never get here
     # (we provide fallbacks in Xclients as well) it can't hurt.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
             netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

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

Notes

1. Fate attenzione che il bit *suid* impostato su file binari (eseguibili) può aprire falle di sicurezza e che il bit *suid* non ha alcun effetto sugli script di shell.
2. Nei moderni sistemi UNIX, lo sticky bit viene utilizzato solo sulle directory e non sui file.
3. Come sottolinea S.C., in una verifica composta, il quoting di una variabile stringa può non essere sufficiente. **`[ -n "$stringa" -o "$a" = "$b" ]`** potrebbe, con alcune versioni di Bash, provocare un errore se `$stringa` fosse vuota. Il modo per evitarlo è quello di aggiungere un carattere extra alle variabili che potrebbero essere vuote, **`[ "x$stringa" != x -o "x$a" = "x$b" ]`** (le "x" si annullano).

Chapter 8. Operazioni ed argomenti correlati

8.1. Operatori

assegnamento

assegnamento di variabile

Inizializzare o cambiare il valore di una variabile

=

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

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

Caution

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

```
#    = come operatore di verifica

if [ "$stringa1" = "$stringa2" ]
# if [ "X$stringa1" = "X$stringa2" ] è più sicuro, evita un
#+ messaggio d'errore se una delle variabili dovesse essere vuota.
# (Le due "X" anteposte si annullano).
then
   comando
fi
```

operatori aritmetici

+

più

-

meno

*

moltiplicazione

/

divisione

**

elevamento a potenza

```
# La versione 2.02 di Bash ha introdotto l’operatore di elevamento a potenza  "**".

let "z=5**3"
echo "z = $z"   # z = 125
```

%

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

```
bash$ echo ‘expr 5 % 3‘
2
```

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

Example 8-1. Massimo comun divisore

```
#!/bin/bash
# gcd.sh: massimo comun divisore
#         Uso dell’algoritmo di Euclide

#  Il "massimo comun divisore" (MCD) di due interi è l’intero
#+ più grande che divide esattamente entrambi.

#  L’algoritmo di Euclide è si basa su divisioni successive.
#  Ad ogni passaggio,
#+ dividendo <---  divisore
#+ divisore  <---  resto
#+ finché resto = 0.
#+ Nell’ultimo passaggio MCD = dividendo.
#
#  Per un’eccellente disamina dell’algoritmo di Euclide, vedi
#  al sito di Jim Loy, http://www.jimloy.com/number/euclids.htm.


# ------------------------------------------------------
# Verifica degli argomenti
ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ]
then
  echo "Utilizzo: ‘basename $0‘ primo-numero secondo-numero"
  exit $E_ERR_ARG
fi
# ------------------------------------------------------
```

```
mcd ()
{

                                #  Assegnamento arbitrario.
  dividendo=$1                  #  Non ha importanza
  divisore=$2                   #+ quale dei due è maggiore.
                                #  Perché?

  resto=1                       #  Se la variabile usata in un ciclo non è
                                #+ inizializzata, il risultato è un errore
 #+ al primo passaggio nel ciclo.

  until [ "$resto" -eq 0 ]
  do
    let "resto = $dividendo % $divisore"
    dividendo=$divisore         # Ora viene ripetuto con 2 numeri più piccoli.
    divisore=$resto
  done                          # Algoritmo di Euclide

}                               # L'ultimo $dividendo è il MCD.


mcd $1 $2

echo; echo "MCD di $1 e $2 = $dividendo"; echo


# Esercizio :
# --------
#  Verificate gli argomenti da riga di comando per essere certi che siano
#+ degli interi, se non lo fossero uscite dallo script con un adeguato
#+ messaggio d'errore.

exit 0
```

`+=`

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

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

`-=`

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

*=

“per-uguale” (moltiplica una variabile per una costante)

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

/=

“diviso-uguale” (divide una variabile per una costante)

%=

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

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

Example 8-2. Utilizzo delle operazioni aritmetiche

```
#!/bin/bash
# Contare fino a 6 in 5 modi diversi.

n=1; echo -n "$n "

let "n = $n + 1"   # Va bene anche let "n = n + 1".
echo -n "$n "

: $((n = $n + 1))
#  I ":" sono necessari perché altrimenti Bash tenta
#+ di interpretare "$((n = $n + 1))" come un comando.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  I ":" sono necessari perché altrimenti Bash tenta
#+ di interpretare "$[ n = $n + 1 ]" come un comando.
# Funziona anche se "n" fosse inizializzata come stringa.
echo -n "$n "

n=$[ $n + 1 ]
#  Funziona anche se "n" fosse inizializzata come stringa.
#* Evitate questo costrutto perché è obsoleto e non
portabile.
echo -n "$n "; echo

# Grazie, Stephane Chazelas.

exit 0
```

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

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

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

Caution

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

```
a=1.5

let "b = $a + 1.3"  # Errore.
#  t2.sh: let: b = 1.5 + 1.3: syntax error in expression
#+ (error token is ".5 + 1.3")

echo "b = $b"       # b=1
```

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

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

operatori bitwise

<<

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

<<=

"scorrimento a sinistra-uguale"

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

>>

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

>>=

“scorrimento a destra-uguale” (inverso di <<=)

&

AND bitwise

&=

“AND bitwise-uguale”

|

OR bitwise

|=

“OR bitwise-uguale”

~

complemento bitwise

!

NOT bitwise

^

XOR bitwise

^=

“XOR bitwise-uguale”

operatori logici

&&

and (logico)

```
if [ $condizione1 ] && [ $condizione2 ]
# Stesso di:  if [ $condizione1 -a $condizione2 ]
# Restituisce vero se entrambe, condizione1 e condizione2, sono vere...

if [[ $condizione1 && $condizione2 ]]    # Funziona anche così.
# Notate che l’operatore && non è consentito nel costrutto [ ... ].
```

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

||

or (logico)

```
if [ $condizione1 ] || [ $condizione2 ]
# Uguale a:  if [ $condizione1 -o $condizione2 ]
# Restituisce vero se o condizione1 o condizione2 è vera...

if [[ $condizione1 || $condizione2 ]]    # Funziona anche così.
# Notate che l'operatore || non è consentito nel costrutto [ ... ].
```

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

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

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Verifica nr.1 eseguita con successo."
else
  echo "Verifica nr.1 fallita."
fi

# ERRORE:  if [ "$a" -eq 24 && "$b" -eq 47 ]
#          cerca di eseguire  ' [ "$a" -eq 24 '
#          e fallisce nella ricerca di corrispondenza di ']'.
#
#    if [[ $a -eq 24 && $b -eq 24 ]]   funziona
#    ("&&" ha un significato diverso nella riga 17 di quello della riga 6.).
#    Grazie, Stephane Chazelas.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Verifica nr.2 eseguita con successo."
else
  echo "Verifica nr.2 fallita."
fi


#  Le opzioni -a e -o offrono
#+ una condizione di verifica composta alternativa.
#  Grazie a Patrick Callahan per la precisazione.


if [ "$a" -eq 24 -a "$b" -eq 47 ]
```

```
then
  echo "Verifica nr.3 eseguita con successo."
else
  echo "Verifica nr.3 fallita."
fi


if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Verifica nr.4 eseguita con successo."
else
  echo "Verifica nr.4 fallita."
fi


a=rinoceronte
b=coccodrillo
if [ "$a" = rinoceronte ] && [ "$b" = coccodrillo ]
then
  echo "Verifica nr.5 eseguita con successo."
else
  echo "Verifica nr.5 fallita."
fi

exit 0
```

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

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

operatori diversi

,

operatore virgola

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

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"              # t1 = 11

let "t2 = ((a = 9, 15 / 3))"  # Imposta "a" e calcola "t2"
echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

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

8.2. Costanti numeriche

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

Example 8-4. Rappresentazione di costanti numeriche

```
#!/bin/bash
# numbers.sh: Rappresentazione di numeri con basi differenti.

# Decimale: quella preimpostata
let "dec = 32"
echo "numero decimale = $dec"            # 32
# Qui non c'è niente di insolito.


# Ottale: numeri preceduti da '0' (zero)
let "oct = 032"
echo "numero ottale = $ott"              # 26
# Risultato visualizzato come decimale.
# --------- ------ -- ----------------

# Esadecimale: numeri preceduti da '0x' o '0X'
let "esa = 0x32"
echo "numero esadecimale = $esa"         # 50
# Risultato visualizzato come decimale.

#  Altre basi: BASE#NUMERO
#  BASE tra 2 e 64.
#  NUMERO deve essere formato dai simboli nell'intervallo indicato da
#+ BASE, vedi di seguito.

let "bin = 2#111100111001101"
echo "numero binario = $bin"             # 31181

let "b32 = 32#77"
echo "numero in base 32 = $b32"          # 231

let "b64 = 64#@_"
echo "numero in base 64 = $b64"          # 4094
#
# Questa notazione funziona solo per un intervallo limitato (2 - 64)
# 10 cifre + 26 caratteri minuscoli + 26 caratteri maiuscoli + @ + _

echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                         # 1295 170 44822 3375


#  Nota importante:
```

```
#  --------------
#  Utilizzare un simbolo al di fuori dell’intervallo della base specificata
#+ provoca un messaggio d’errore.

let "ott_errato = 081"
#  numbers.sh: let: ott_errato = 081: value too great for base
#+ (error token is "081")
#             I numeri ottali utilizzano solo cifre nell’intervallo 0 - 7.

exit 0       # Grazie, Rich Bartell e Stephane Chazelas, per il chiarimento.
```

Part 3. Oltre i fondamenti

Chapter 9. Variabili riviste

Utilizzate in modo appropriato, le variabili possono aumentare la potenza e la flessibilità degli script. Per questo è necessario conoscere tutte le loro sfumature e sottigliezze.

9.1. Variabili interne

Variabili builtin (incorporate)

sono quelle variabili che determinano il comportamento dello script Bash

`$BASH`

il percorso dell'eseguibile *Bash*

```
bash$ echo $BASH
/bin/bash
```

`$BASH_ENV`

variabile d'ambiente che punta al file di avvio di Bash, che deve essere letto quando si invoca uno script

`$BASH_VERSINFO[n]`

un array di 6 elementi contenente informazioni sulla versione Bash installata. È simile a `$BASH_VERSION`, vedi oltre, ma più dettagliata.

```
# Informazioni sulla versione Bash:

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 2                      # nr. della major version.
# BASH_VERSINFO[1] = 05                     # nr. della minor version.
# BASH_VERSINFO[2] = 8                      # nr. del patch level.
# BASH_VERSINFO[3] = 1                      # nr. della build version.
# BASH_VERSINFO[4] = release                # Stato della release.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architettura.
                                            # (uguale a $MACHTYPE).
```

`$BASH_VERSION`

la versione Bash installata

```
bash$ echo $BASH_VERSION
2.04.12(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

Un buon metodo per determinare quale shell è in funzione è verificare $BASH_VERSION. $SHELL potrebbe non fornire necessariamente una risposta corretta.

`$DIRSTACK`

il contenuto della locazione più alta dello stack delle directory (determinato da pushd e popd)

Questa variabile corrisponde al comando dirs, tuttavia **dirs** mostra l'intero contenuto dello stack delle directory.

`$EDITOR`

l'editor di testo predefinito invocato da uno script, solitamente **vi** o **emacs**.

`$EUID`

numero ID "effettivo" dell'utente

Numero identificativo corrispondente all'identità effettiva assunta dall'utente corrente, solitamente modificata tramite il comando su.

Caution

`$EUID`, di conseguenza, non è necessariamente uguale a $UID.

`$FUNCNAME`

nome della funzione corrente

```
xyz23 ()
{
  echo "$FUNCNAME è in esecuzione."  # xyz23 è in esecuzione.
}

xyz23

echo "NOME FUNZIONE = $FUNCNAME"     # NOME FUNZIONE =
                                     # Valore nullo all'esterno della funzione.
```

`$GLOBIGNORE`

un elenco di nomi di file da escludere dalla ricerca nel globbing

`$GROUPS`

i gruppi a cui appartiene l'utente corrente

È l'elenco (array) dei numeri id dei gruppi a cui appartiene l'utente corrente, così come sono registrati nel file `/etc/passwd`.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1


root# echo ${GROUPS[5]}
6
```

`$HOME`

directory home dell'utente, di solito `/home/nomeutente` (vedi Example 9-13)

`$HOSTNAME`

Il comando hostname assegna il nome del sistema, durante il boot in uno script init. Tuttavia è la funzione `gethostname()` che imposta la variabile interna Bash `$HOSTNAME`. Vedi anche Example 9-13.

`$HOSTTYPE`

tipo di macchina

Come $MACHTYPE, identifica il sistema hardware, ma in forma ridotta.

```
bash$ echo $HOSTTYPE
i686
```

`$IFS`

separatore di campo (input field separator)

Il valore prestabilito è una spaziatura (spazio, tabulazione e ritorno a capo), ma può essere modificato, per esempio, per verificare un file dati che usa la virgola come separatore di campi. E' da notare che $* utilizza il primo carattere contenuto in `$IFS`. Vedi Example 5-1.

```
bash$ echo $IFS | cat -vte
$


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```

Caution

`$IFS` non tratta la spaziatura allo stesso modo degli altri caratteri.

Example 9-1. $IFS e gli spazi

```
#!/bin/bash
# $IFS gestisce gli spazi in modo diverso dagli altri caratteri.

output_arg_uno_per_riga()
{
  for arg
  do echo "[$arg]"
  done
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
output_arg_uno_per_riga $var   # output_arg_uno_per_riga ‘echo " a  b c   "‘
#
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
var=":a::b:c:::"               # Come prima, ma con ":" anziché " ".
output_arg_uno_per_riga $var
#
# []
# [a]
# []
# [b]
# [c]
# []
# []
# []

# In awk si ottiene lo stesso risultato con il separatore di campo "FS".

# Grazie, Stephane Chazelas.

echo

exit 0
```

(Grazie, S. C., per i chiarimenti e gli esempi.)

`$IGNOREEOF`

ignora EOF: quanti end-of-file (control-D) la shell deve ignorare prima del logout

`$LC_COLLATE`

Spesso impostata nei file `.bashrc` o `/etc/profile`, questa variabile controlla l'ordine di collazione nell'espansione del nome del file e nella ricerca di corrispondenza. Se mal gestita, `LC_COLLATE` può provocare risultati inattesi nel globbing dei nomi dei file.

> **Note:** Dalla versione 2.05 di Bash, il globbing dei nomi dei file non fa più distinzione tra lettere minuscole e maiuscole, in un intervallo di caratteri specificato tra parentesi quadre. Per esempio, **ls [A-M]*** restituisce sia `File1.txt` che `file1.txt`. Per riportare il globbing all'abituale comportamento, si imposti `LC_COLLATE` a `C` con **`export LC_COLLATE=C`** nel file `/etc/profile` e/o `~/.bashrc`.

`$LC_CTYPE`

Questa variabile interna controlla l'interpretazione dei caratteri nel globbing e nella ricerca di corrispondenza (modello).

`$LINENO`

Questa variabile contiene il numero della riga dello script di shell in cui essa appare. Ha valore solo nello script in cui si trova. È utile in modo particolare nel debugging.

```
# *** INIZIO BLOCCO DI DEBUG ***
ultimo_cmd_arg=$_  # Viene salvato.

echo "Alla riga numero $LINENO, variabile \"v1\" = $v1"
echo "Ultimo argomento eseguito = $ultimo_cmd_arg"
# *** FINE BLOCCO DI DEBUG ***
```

`$MACHTYPE`

tipo di macchina

Identifica il sistema hardware in modo dettagliato.

```
bash$ echo $MACHTYPE
i486-slackware-linux-gnu
```

`$OLDPWD`

precedente directory di lavoro ("OLD-print-working-directory", la directory precedente in cui vi trovavate, prima dell'ultimo comando cd)

`$OSTYPE`

nome del sistema operativo

```
bash$ echo $OSTYPE
linux
```

`$PATH`

i percorsi delle directory in cui si trovano i file eseguibili (binari), di solito `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, etc.

Quando viene dato un comando, la shell ricerca automaticamente il *percorso* dell'eseguibile. Questo è possibile perché tale percorso è memorizzato nella variabile d'ambiente, `$PATH`, che è un elenco di percorsi possibili separati da : (due punti). Di solito il sistema conserva la configurazione di `$PATH` nel file `/etc/profile` e/o `~/.bashrc` (vedi Chapter 27).

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

`PATH=${PATH}:/opt/bin` aggiunge la directory `/opt/bin` ai percorsi predefiniti. Usato in uno script rappresenta un espediente per aggiungere temporaneamente una directory a `$PATH`. Quando lo script termina viene ripristinato il valore originale di `$PATH` (questo perché un processo figlio, qual'è uno script, non può modificare l'ambiente del processo genitore, la shell).

> **Note:** La "directory di lavoro", `./`, di solito per ragioni di sicurezza, non è compresa in `$PATH`.

`$PIPESTATUS`

Exit status dell'ultima pipe in *foreground* (primo piano). È piuttosto interessante in quanto non fornisce lo stesso exit status dell'ultimo comando eseguito.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | comando_errato
bash: comando_errato: command not found
bash$ echo $PIPESTATUS
141

bash$ ls -al | comando_errato
bash: comando_errato: command not found
bash$ echo $?
127
```

Caution

La variabile `$PIPESTATUS`, in una shell di login, potrebbe contenere un errato valore 0.

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
```

I comandi precedenti, eseguiti in uno script, avrebbero prodotto il risultato atteso `0 1 0`.

Grazie a Wayne Pollock per la puntualizzazione e per aver fornito l'esempio precedente.

`$PPID`

Il `$PPID` di un processo non è che l'ID di processo (`pid`) del processo genitore. [1]

Lo si confronti con il comando pidof.

`$PROMPT_COMMAND`

Variabile che contiene un comando che deve essere eseguito immediatamente prima della visualizzazione del prompt primario, `$PS1` .

`$PS1`

È il prompt principale, quello che compare sulla riga di comando.

`$PS2`

Prompt secondario. Compare quando è atteso un ulteriore input (il comando non è ancora terminato). Viene visualizzato come ">".

`$PS3`

Prompt di terzo livello, visualizzato in un ciclo select (vedi Example 10-29).

`$PS4`

Prompt di quarto livello. Viene visualizzato all'inizio di ogni riga di output quando lo script è stato invocato con l'opzione -x. Viene visualizzato come "+".

`$PWD`

Directory di lavoro (directory corrente)

È analoga al comando builtin pwd.

```
#!/bin/bash

E_ERRATA_DIRECTORY=73

clear # Pulisce lo schermo.

DirectoryDestinazione=/home/bozo/projects/GreatAmericanNovel

cd $DirectoryDestinazione
echo "Cancellazione dei vecchi file in $DirectoryDestinazione."

if [ "$PWD" != "$DirectoryDestinazione" ]
then    # Evita di cancellare per errore una directory sbagliata.
  echo "Directory errata!"
  echo "Sei in $PWD, non in $DirectoryDestinazione!"
  echo "Salvo!"
  exit $E_ERRATA_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*    # Cancella i file i cui nomi iniziano con un punto.
# rm -f .[^.]* ..?*   per cancellare file che iniziano con due o più punti.
# (shopt -s dotglob; rm -f *)   anche in questo modo.
# Grazie, S.C. per la puntualizzazione.

#  I nomi dei file possono essere formati da tutti i caratteri nell'intervallo
#+ 0 - 255, tranne "/". La cancellazione di file che iniziano con caratteri
#+ inconsueti è lasciata come esercizio.

# Altre eventuali operazioni.

echo
echo "Fatto."
echo "Cancellati i vecchi file in $DirectoryDestinazione."
echo


exit 0
```

`$REPLY`

È la variabile preimpostata quando non ne viene fornita alcuna a read. È utilizzabile anche con i menu select. In questo caso, però, fornisce solo il numero che indica la variabile scelta, non il valore della variabile.

```
#!/bin/bash
# reply.sh

# REPLY è il valore preimpostato per il comando 'read'.

echo
```

```
    echo -n "Qual'è la tua verdura preferita?"
    read

    echo "La tua verdura preferita è $REPLY."
    #  REPLY contiene il valore dell'ultimo  "read" se e solo se
    #+ non è stata indicata alcuna variabile.

    echo
    echo -n "Qual'è il tuo frutto preferito?"
    read frutto
    echo "Il tuo frutto preferito è $frutto."
    echo "ma..."
    echo "Il valore di \$REPLY è ancora $REPLY."
    #  $REPLY è ancora impostato al valore precedente perché
    #+ la variabile $frutto contiene il nuovo valore letto con "read".

    echo

    exit 0
```

`$SECONDS`

Numero di secondi trascorsi dall'inizio dell'esecuzione dello script.

```
    #!/bin/bash

    TEMPO_LIMITE=10
    INTERVALLO=1

    echo
    echo "Premi Control-C per terminare prima di $TEMPO_LIMITE secondi."
    echo

    while [ "$SECONDS" -le "$TEMPO_LIMITE" ]
    do
      if [ "$SECONDS" -eq 1 ]
      then
        unita=secondo
      else
        unita=secondi
      fi

      echo "Questo script è in esecuzione da $SECONDS $unita."
      #  Su una macchina lenta o sovraccarica, lo script potrebbe saltare un
      #+ conteggio una volta ogni tanto.
      sleep $INTERVALLO
    done

    echo -e "\a"  # Beep!

    exit 0
```

`$SHELLOPTS`

l’elenco delle opzioni di shell abilitate. È una variabile in sola lettura

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

`$SHLVL`

Livello della shell. Profondità di annidamento di Bash. Se, da riga di comando, $SHLVL vale 1, in uno script questo valore viene aumentato a 2.

`$TMOUT`

Se la variabile d’ambiente *`$TMOUT`* è impostata ad un valore *tempo* diverso da zero, il prompt della shell termina dopo *tempo* secondi. Questo provoca il logout.

Dalla versione Bash 2.05b è possibile utilizzare *`$TMOUT`* negli script in combinazione con read.

```
# Funziona negli script con Bash versione 2.05b e successive.

TMOUT=3    # Imposta il prompt alla durata di tre secondi.

echo "Qual’è la tua canzone preferita?"
echo "Svelto, hai solo $TMOUT secondi per rispondere!"
read canzone

if [ -z "$canzone" ]
then
  canzone="(nessuna risposta)"
  # Risposta preimpostata.
fi

echo "La tua canzone preferita è $canzone."
```

Esistono altri metodi, più complessi, per implementare un input temporizzato in uno script. Una possibile alternativa è quella di impostare un ciclo di temporizzazione per segnalare allo script quando il tempo è scaduto. Ma anche così è necessaria una routine per la gestione di un segnale per catturare (trap) (vedi Example 30-5) l’interrupt generato dal ciclo di temporizzazione (whew!).

Example 9-2. Input temporizzato

```
#!/bin/bash
# timed-input.sh

# TMOUT=3            inutile in uno script

TEMPOLIMITE=3  #  In questo caso tre secondi. Può essere impostato
               #+ ad un valore diverso.
```

```
VisualizzaRisposta()
{
  if [ "$risposta" = TIMEOUT ]
  then
    echo $risposta
  else       # Ho voluto tenere separati i due esempi.
    echo "La tua verdura preferita è $risposta"
    kill $!  #  Uccide la funzione AvvioTimer in esecuzione in
             #+ background perché non più necessaria. $! è il PID
             #+ dell'ultimo job in esecuzione in background.

  fi

}



AvvioTimer()
{
  sleep $TEMPOLIMITE && kill -s 14 $$ &
  # Attende 3 secondi, quindi invia il segnale SIGALARM allo script.
}

Int14Vettore()
{
  risposta="TIMEOUT"
  VisualizzaRisposta
  exit 14
}

trap Int14Vettore 14   # Interrupt del timer (14) modificato allo scopo.

echo "Qual'è la tua verdura preferita? "
AvvioTimer
read risposta
VisualizzaRisposta

#  Ammettiamolo, questa è un'implementazione tortuosa per temporizzare
#+ l'input, comunque l'opzione "-t" di "read" semplifica il compito.
#  Vedi sopra "t-out.sh".

#  Se desiderate qualcosa di più elegante... prendete in considerazione
#+ la possibilità di scrivere l'applicazione in C o C++,
#+ utilizzando le funzioni di libreria appropriate, come 'alarm' e 'setitimer'.

exit 0
```

Un'alternativa è l'utilizzo di stty.

Example 9-3. Input temporizzato, un ulteriore esempio

```
#!/bin/bash
# timeout.sh

# Scritto da Stephane Chazelas e modificato dall'autore.

INTERVALLO=5                # intervallo di timeout

leggi_temporizzazione() {
  timeout=$1
  nomevar=$2
  precedenti_impostazioni_tty=‘stty -g‘
  stty -icanon min 0 time ${timeout}0
  eval read $nomevar      # o semplicemente    read $nomevar
  stty "$precedenti_impostazioni_tty"
  # Vedi la pagina di manuale di "stty".
}

echo; echo -n "Come ti chiami? Presto! "
leggi_temporizzazione $INTERVALLO nome

# Questo potrebbe non funzionare su tutti i tipi di terminale.
# Il timeout massimo, infatti, dipende dal terminale.
# (spesso è di 25.5 secondi).

echo

if [ ! -z "$nome" ]  # se il nome è stato immesso prima del timeout...
then
  echo "Ti chiami $nome."
else
  echo "Tempo scaduto."
fi

echo

# Il comportamento di questo script è un po' diverso da "timed-input.sh".
# Ad ogni pressione di tasto, la temporizzazione ricomincia da capo.

exit 0
```

Forse, il metodo più semplice è quello di usare read con l'opzione `-t`.

Example 9-4. read temporizzato

```
#!/bin/bash
# t-out.sh
# Ispirato da un suggerimento di "syngin seven" (grazie).

TEMPOLIMITE=4        # 4 secondi
```

```
read -t $TEMPOLIMITE variabile <&1

echo

if [ -z "$variabile" ]
then
  echo "Tempo scaduto, la variabile non è stata impostata."
else
  echo "variabile = $variabile"
fi

exit 0

# Esercizio per il lettore:
# ------------------------
# Perché è necessaria la redirezione (<&1) alla riga 8?
# Cosa succede se viene omessa?
```

`$UID`

numero ID dell'utente

è il numero identificativo dell'utente corrente com'è registrato nel file `/etc/passwd`.

Rappresenta l'id reale dell'utente, anche nel caso abbia assunto temporaneamente un'altra identità per mezzo di su. `$UID` è una variabile in sola lettura e non può essere modificata né da riga di comando né in uno script. È il sostituto del builtin id.

Example 9-5. Sono root?

```
#!/bin/bash
# am-i-root.sh:   Sono root o no?

ROOT_UID=0   # Root ha $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]  #  Il vero "root" avrà la compiacenza
                               #+ di aspettare?
then
  echo "Sei root."
else
  echo "Sei un utente normale (ma la mamma ti vuol bene lo stesso)."
fi

exit 0


# ============================================================== #
# Il codice seguente non viene eseguito perché lo script è già terminato.

# Un metodo alternativo per andare al fondo della questione:

NOME_ROOT=root
```

```
nomeutente=‘id -nu‘              # Oppure...   nomeutente=‘whoami‘
if [ "$nomeutente" = "$NOME_ROOT" ]
then
  echo "Rooty, toot, toot. Sei root."
else
  echo "Sei solo un semplice utente."
fi
```

Vedi anche Example 2-2.

> **Note:** Le variabili `$ENV`, `$LOGNAME`, `$MAIL`, `$TERM`, `$USER`, e `$USERNAME` *non* sono builtin di Bash. Vengono, comunque, impostate spesso come variabili d’ambiente in uno dei file di avvio di Bash. `$SHELL`, è il nome della shell di login dell’utente, può essere impostata dal file `/etc/passwd` o da uno script “init”. Anche questa non è un builtin di Bash.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Parametri Posizionali

`$0`, `$1`, `$2`, ecc.

rappresentano i diversi parametri che vengono passati da riga di comando ad uno script, ad una funzione, o per impostare una variabile (vedi Example 4-5 e Example 11-14)

`$#`

numero degli argomenti passati da riga di comando, [2] ovvero numero dei parametri posizionali (vedi Example 34-2)

`$*`

Tutti i parametri posizionali visti come un’unica parola

> **Note:** “`$*`” dev’essere usata con il quoting.

`$@`

Simile a $*, ma ogni parametro è una stringa tra apici (quoting), vale a dire, i parametri vengono passati intatti, senza interpretazione o espansione. Questo significa, tra l'altro, che ogni parametro dell'elenco viene considerato come una singola parola.

> **Note:** Naturalmente, "`$@`" va usata con il quoting.

Example 9-6. arglist: Elenco degli argomenti con $* e $@

```
#!/bin/bash
# arglist.sh
# Invocate lo script con molti argomenti, come "uno due tre".

E_ERR_ARG=65

if [ ! -n "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ argomento1 argomento2 ecc."
  exit $E_ERR_ARG
fi

echo

indice=1          # Inizializza il contatore.

echo "Elenco degli argomenti con \"\$*\":"
for arg in "$*"  # Non funziona correttamente se "$*" non è tra apici.
do
  echo "Argomento nr.$indice = $arg"
  let "indice+=1"
done              # $* vede tutti gli argomenti come un’unica parola.
echo "Tutto l’elenco come parola singola."

echo

indice=1          # Reimposta il contatore.
                  # Cosa succede se vi dimenticate di farlo?

echo "Elenco degli argomenti con \"\$@\":"
for arg in "$@"
do
  echo "Argomento nr.$indice = $arg"
  let "indice+=1"
done              # $@ vede gli argomenti come parole separate.
echo "Elenco composto da diverse parole."

echo
```

```
indice=1         # Reimposta il contatore.

echo "Eleco degli argomenti con \$* (senza quoting):"
for arg in $*
do
  echo "Argomento nr.$indice = $arg"
  let "indice+=1"
done             # $* senza quoting vede gli argomenti come parole separate.
echo "Elenco composto da diverse parole."

exit 0
```

Dopo uno **shift**, venendo a mancare il precedente `$1`, che viene perso, `$@` contiene i restanti parametri posizionali.

```
#!/bin/bash
# Da eseguire con  ./nomescript 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Ogni "shift" perde il precedente $1.
# Come conseguenza "$@" contiene i parametri rimanenti.
```

All'interno degli script di shell, la variabile speciale `$@` viene utilizzata come strumento per filtrare un dato input. Il costrutto **cat "$@"** permette di gestire un input da uno script, dallo `stdin` o da file forniti come parametri. Vedi Example 12-18 e Example 12-19.

> **Caution**
>
> I parametri `$*` e `$@` talvolta si comportano in modo incoerente e sorprendente. Questo dipende dall'impostazione di $IFS.

Example 9-7. Comportamento incoerente di $* e $@

```
#!/bin/bash

#  Comportamento non corretto delle variabili interne Bash "$*" e "$@",
#+ dipendente dal fatto che vengano utilizzate o meno con il "quoting".
#  Gestione incoerente della suddivisione delle parole e del ritorno a capo.

set -- "Il primo" "secondo" "il:terzo" "" "Il: :quinto"
# Imposta gli argomenti dello script, $1, $2, ecc.

echo

echo 'IFS con il valore preimpostato, utilizzando "$*"'
```

```
c=0
for i in "$*"               # tra doppi apici
do echo "$((c+=1)): [$i]"   # Questa riga rimane invariata in tutti gli esempi.
                            # Visualizza gli argomenti.
done
echo ---

echo 'IFS con il valore preimpostato, utilizzando $*'
c=0
for i in $*                 # senza apici
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS con il valore preimpostato, utilizzando "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS con il valore preimpostato, utilizzando $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", utilizzando "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", utilizzando "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando $var (var=$*)'
c=0
```

```
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", utilizzando $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", utilizzando $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilizzando "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", utilizzando "$var" (var="$@")'
c=0
for i in "$var"
```

```
do echo "$((c+=1)): [$i]"
done
echo ---

echo ’IFS=":", utilizzando $var (var="$@")’
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Provate questo script con ksh o zsh -y.

exit 0

# Script d’esempio di Stephane Chazelas,
# con piccole modifiche apportate dall’autore.
```

Note: I parametri **$@** e **$*** differiscono solo quando vengono posti tra doppi apici.

Example 9-8. `$*` e `$@` quando `$IFS` è vuoto

```
#!/bin/bash

#  Se $IFS è impostata, ma vuota, allora "$*" e "$@" non
#+ visualizzano i parametri posizionali come ci si aspetterebbe.

mecho ()       # Visualizza i parametri posizionali.
{
echo "$1,$2,$3";
}


IFS=""         # Impostata, ma vuota.
set a b c      # Parametri posizionali.

mecho "$*"     # abc„
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

#  Il comportamento di $* e $@ quando $IFS è vuota dipende da quale
#+ versione Bash o sh è in esecuzione. È quindi sconsigliabile fare
#+ affidamento su questa "funzionalità" in uno script.


# Grazie, S.C.
```

```
exit 0
```

Altri Parametri Speciali

`$-`

Opzioni passate allo script (utilizzando set). Vedi Example 11-14.

> **Caution**
>
> In origine era un costrutto *ksh* che è stato adottato da Bash, ma, sfortunatamente, non sembra funzionare in modo attendibile negli script Bash. Un suo possibile uso è quello di eseguire un'autoverifica di interattività.

`$!`

PID (ID di processo) dell'ultimo job eseguito in background

```
LOG=$0.log

COMANDO1="sleep 100"

echo "Registra i PID dei comandi in background dello script: $0" >> "$LOG"
# Possono essere così controllati e, se necessario, "uccisi".
echo >> "$LOG"

# Registrazione dei comandi.

echo -n "PID di \"$COMANDO1\":  " >> "$LOG"
${COMANDO1} &
echo $! >> "$LOG"
# PID di "sleep 100":  1506

# Grazie a Jacques Lederer, per il suggerimento.
```

`$_`

Variabile speciale impostata all'ultimo argomento del precedente comando eseguito.

Example 9-9. Variabile underscore

```
#!/bin/bash

echo $_              # /bin/bash
                     # digitate solo /bin/bash per eseguire lo script.

du >/dev/null        # Non viene visualizzato alcun output del comando.
```

```
    echo $_              # du

    ls -al >/dev/null    # Non viene visualizzato alcun output del comando.
    echo $_              # -al  (ultimo argomento)

    :
    echo $_              # :
```

`$?`

Exit status di un comando, funzione, o dello stesso script (vedi Example 23-3)

`$$`

ID di processo dello script. La variabile `$$` viene spesso usata negli script per creare un nome di file temporaneo “unico” (vedi Example A-14, Example 30-6, Example 12-24 e Example 11-24). Di solito è più semplice che invocare mktemp.

9.2. Manipolazione di stringhe

Bash supporta un numero sorprendentemente elevato di operazioni per la manipolazione delle stringhe. Purtroppo, questi strumenti mancano di organizzazione e razionalizzazione. Alcuni sono un sotto insieme della sostituzione di parametro, altri appartengono alle funzionalità del comando UNIX expr. Tutto questo si traduce in una sintassi dei comandi incoerente ed in una sovrapposizione di funzionalità, per non parlare della confusione.

Lunghezza della Stringa

${#stringa}

expr length $stringa

expr "$stringa" : ’.*’

```
    stringZ=abcABC123ABCabc

    echo ${#stringZ}                 # 15
    echo ‘expr length $stringZ‘      # 15
    echo ‘expr "$stringZ" : ’.*’‘    # 15
```

Example 9-10. Inserire una riga bianca tra i paragrafi in un file di testo

```
#!/bin/bash
# paragraph-space.sh
```

```
#  Inserisce una riga bianca tra i paragrafi di un file di testo con
#+ spaziatura semplice.
#  Utilizzo: $0 <NOMEFILE

LUNMIN=45        # Potrebbe rendersi necessario modificare questo valore.
#  Si assume che le righe di lunghezza inferiore a $LUNMIN caratteri
#+ siano le ultime dei paragrafi.

while read riga  # Per tutte le righe del file di input...
do
  echo "$riga"   # Visualizza la riga.

  len=${#riga}
  if [ "$len" -lt "$LUNMIN" ]
    then echo    # Aggiunge la riga bianca.
  fi
done

exit 0
```

Lunghezza della sottostringa verificata nella parte iniziale della stringa

expr match "$stringa" ’$sottostringa’

`$sottostringa` è un’espressione regolare.

expr "$stringa" : ’$sottostringa’

`$sottostringa` è un’espressione regolare.

```
stringZ=abcABC123ABCabc
#       |------|

echo ‘expr match "$stringZ" ’abc[A-Z]*.2’’   # 8
echo ‘expr "$stringZ" : ’abc[A-Z]*.2’’       # 8
```

Indice

expr index $stringa $sottostringa

Posizione numerica in $stringa del primo carattere compreso in $sottostringa che è stato verificato.

```
stringZ=abcABC123ABCabc
echo ‘expr index "$stringZ" C12‘             # 6
                                             # Posizione di C.

echo ‘expr index "$stringZ" 1c‘              # 3
# ’c’ (in terza posizione) viene verificato prima di ’1’.
```

È quasi uguale alla funzione *strchr()* del C.

Estrazione di sottostringa

${stringa:posizione}

Estrae la sottostringa da *$stringa* iniziando da *$posizione*.

Se il parametro `$stringa` è "*" o "@", allora vengono estratti i parametri posizionali, [3] iniziando da `$posizione`.

${stringa:posizione:lunghezza}

Estrae una sottostringa di *$lunghezza* caratteri da *$stringa* iniziando da *$posizione*.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       L'indicizzazione inizia da 0.

echo ${stringZ:0}                           # abcABC123ABCabc
echo ${stringZ:1}                           # bcABC123ABCabc
echo ${stringZ:7}                           # 23ABCabc

echo ${stringZ:7:3}                         # 23A
                                            # Sottostringa di tre caratteri.



# È possibile indicizzare partendo dalla fine della stringa?

echo ${stringZ:-4}                          # abcABC123ABCabc
# Restituisce l'intera stringa, come con ${parametro:-default}.
# Tuttavia . . .

echo ${stringZ:(-4)}                        # Cabc
echo ${stringZ: -4}                         # Cabc
# Ora funziona.
# Le parentesi, o l'aggiunta di uno spazio, "preservano" il parametro negativo.

# Grazie, Dan Jacobson, per averlo evidenziato.
```

Se il parametro `$stringa` è "*" o "@", vengono estratti un massimo di `$lunghezza` parametri posizionali, iniziando da `$posizione`.

```
echo ${*:2}          #  Visualizza tutti i parametri iniziando dal secondo.
echo ${@:2}          #  Come prima.

echo ${*:2:3}        #  Visualizza tre parametri posizionali
                     #+ iniziando dal secondo.
```

expr substr $stringa $posizione $lunghezza

Estrae *$lunghezza* caratteri da *$stringa* iniziando da *$posizione*.

```
stringZ=abcABC123ABCabc
#       123456789......
#       L'indicizzazione inizia da 1.

echo 'expr substr $stringZ 1 2'              # ab
echo 'expr substr $stringZ 4 3'              # ABC
```

expr match "$stringa" '\($sottostringa\)'

Estrae *$sottostringa* dalla parte iniziale di *$stringa*, dove *$sottostringa* è un'espressione regolare.

expr "$stringa" : '\($sottostringa\)'

Estrae *$sottostringa* dalla parte iniziale di *$stringa*, dove *$sottostringa* è un'espressione regolare.

```
stringZ=abcABC123ABCabc
#       =======

echo 'expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)''   # abcABC1
echo 'expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)''       # abcABC1
echo 'expr "$stringZ" : '\(.......\)''                   # abcABC1
# Tutte le forme precedenti danno lo stesso risultato.
```

expr match "$stringa" '.*\($sottostringa\)'

Estrae *$sottostringa* dalla parte *finale* di *$stringa*, dove *$sottostringa* è un'espressione regolare.

expr "$stringa" : '.*\($sottostringa\)'

Estrae *$sottostringa* dalla parte *finale* di *$stringa*, dove *$sottostringa* è un'espressione regolare.

```
stringZ=abcABC123ABCabc
#               ======

echo 'expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)''    # ABCabc
echo 'expr "$stringZ" : '.*\(......\)''                      # ABCabc
```

Rimozione di sottostringa

${stringa#sottostringa}

Toglie l'occorrenza più breve di *`$sottostringa`* dalla parte *iniziale* di *`$stringa`*.

${stringa##sottostringa}

Toglie l'occorrenza più lunga di *`$sottostringa`* dalla parte *iniziale* di *`$stringa`*.

```
stringZ=abcABC123ABCabc
#       |----|
#       |----------|

echo ${stringZ#a*C}      # 123ABCabc
# È stata tolta l'occorrenza più breve compresa tra 'a' e 'C'.

echo ${stringZ##a*C}     # abc
# È stata tolta l'occorrenza più lunga compresa tra 'a' e 'C'.
```

${stringa%sottostringa}

Toglie l'occorrenza più breve di *`$sottostringa`* dalla parte *finale* di *`$stringa`*.

${stringa%%sottostringa}

Toglie l'occorrenza più lunga di *`$sottostringa`* dalla parte *finale* di *`$stringa`*.

```
stringZ=abcABC123ABCabc
#                    ||
#        |------------|

echo ${stringZ%b*c}      # abcABC123ABCa
#  È stata tolta l'occorrenza più breve compresa
#+ tra 'b' e 'c', dalla fine di $stringZ.

echo ${stringZ%%b*c}     # a
#  È stata tolta l'occorrenza più lunga compresa
#+ tra 'b' e 'c', dalla fine di $stringZ.
```

Example 9-11. Conversione di formato di file grafici e modifica del nome dei file

```
#!/bin/bash
#  cvt.sh:
#  Converte tutti i file immagine MacPaint, in una directory data,
#+ nel formato "pbm".
#  Viene utilizzato l'eseguibile "macptopbm" del pacchetto "netpbm",
#+ mantenuto da Brian Henderson (bryanh@giraffe-data.com). Netpbm di
#+ solito è compreso nella maggior parte delle distribuzioni standard Linux.
```

```
OPERAZIONE=macptopbm
ESTENSIONE=pbm          # Nuova estensione dei nomi dei file.

if [ -n "$1" ]
then
  directory=$1      #  Se viene fornito il nome di una directory come
                    #+ argomento dello script...
else
  directory=$PWD    # Altrimenti viene utilizzata la directory corrente.
fi

#  Si assume che tutti i file immagine nella directory siano dei MacPaint,
# + con estensione ".mac".

for file in $directory/*    # Globbing dei nomi dei file.
do
  nomefile=${file%.*c}      #  Toglie l'estensione ".mac" dal nome del file
                            #+ (.*c' verifica tutto tra '.' e 'c', compresi).
  $OPERAZIONE $file > "$nomefile.$ESTENSIONE"
                            #  Converte e redirige il file con una nuova
                            #+ estensione.
  rm -f $file               #  Cancella i file originali dopo la conversione.
  echo "$nomefile.$ESTENSIONE"  # Visualizza quello che avviene allo stdout.
done

exit 0

# Esercizio:
# ----------
#  Così com'è, lo script converte "tutti" i file presenti nella
#+ directory di lavoro corrente.
#  Modificatelo in modo che agisca "solo" sui file con estensione ".mac".
```

Sostituzione di sottostringa

${stringa/sottostringa/sostituto}

Sostituisce la prima occorrenza di *`$sottostringa`* con *`$sostituto`*.

${stringa//sottostringa/sostituto}

Sostituisce tutte le occorrenze di *`$sottostringa`* con *`$sostituto`*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}   # xyzABC123ABCabc
                          # Sostituisce la prima occorrenza di 'abc' con 'xyz'.

echo ${stringZ//abc/xyz}  # xyzABC123ABCxyz
                          # Sostituisce tutte le occorrenze di 'abc' con 'xyz'.
```

${stringa/#sottostringa/sostituto}

Se *`$sottostringa`* viene verificata all'*inizio* di *`$stringa`*, allora *`$sostituto`* rimpiazza *`$sottostringa`*.

${stringa/%sottostringa/sostituto}

Se *`$sottostringa`* viene verificata alla *fine* di *`$stringa`*, allora *`$sostituto`* rimpiazza *`$sottostringa`*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}     # XYZABC123ABCabc
                             # Sostituisce l'occorrenza iniziale 'abc' 'XYZ'.

echo ${stringZ/%abc/XYZ}     # abcABC123ABCXYZ
                             # Sostituisce l'occorrenza finale 'abc' con 'XYZ'.
```

9.2.1. Manipolare stringhe con awk

Uno script Bash può ricorrere alle capacità di manipolazione delle stringhe di awk, come alternativa all'utilizzo dei propri operatori builtin.

Example 9-12. Modi alternativi di estrarre sottostringhe

```
#!/bin/bash
# substring-extraction.sh

Stringa=23skidoo1
#       012345678    Bash
#       123456789    awk
# Fate attenzione al diverso sistema di indicizzazione della stringa:
# Bash numera il primo carattere della stringa con '0'.
# Awk  numera il primo carattere della stringa con '1'.

echo ${Stringa:2:4} # posizione 3 (0-1-2), 4 caratteri di lunghezza
                                        # skid

#  L'equivalente awk di ${stringa:pos:lunghezza} è
#+ substr(stringa,pos,lunghezza).
echo | awk '
{ print substr("'"${Stringa}"'",3,4)     # skid
}
'
#  Collegando ad awk un semplice comando "echo" gli viene dato un
#+ input posticcio, in questo modo non diventa più necessario
#+ fornirgli il nome di un file.
```

```
exit 0
```

9.2.2. Ulteriori approfondimenti

Per altro materiale sulla manipolazione delle stringhe negli script, si faccia riferimento alSection 9.3 e alla sezione attinente all'elenco dei comandi expr. Per gli script d'esempio, si veda:

1. Example 12-7
2. Example 9-15
3. Example 9-16
4. Example 9-17
5. Example 9-19

9.3. Sostituzione di parametro

Manipolare e/o espandere le variabili

`${parametro}`

Uguale a *`$parametro`*, cioè, valore della variabile *`parametro`*. In alcuni contesti funziona solo la forma meno ambigua, *`${parametro}`*.

Può essere utilizzato per concatenare delle stringhe alle variabili.

```
tuo_id=${USER}-su-${HOSTNAME}
echo "$tuo_id"
#
echo "Vecchio \$PATH = $PATH"
PATH=${PATH}:/opt/bin  # Aggiunge /opt/bin a $PATH per la durata dello script.
echo "Nuovo \$PATH = $PATH"
```

`${parametro-default}`
`${parametro:-default}`

Se parametro non è impostato, viene impostato al valore fornito da default.

```
echo ${nomeutente-‘whoami‘}
#  Visualizza il risultato del comando ‘whoami‘, se la variabile
#+ $nomeutente non è ancora impostata.
```

Note: *`${parametro-default}`* e *`${parametro:-default}`* sono quasi uguali. L'aggiunta dei : serve solo quando *parametro* è stato dichiarato, ma non impostato.

```
#!/bin/bash

nomeutente0=
# nomeutente0 è stato dichiarato, ma contiene un valore nullo.
echo "nomeutente0 = ${nomeutente0-‘whoami‘}"
# Non visualizza niente.

echo "nomeutente1 = ${nomeutente1-‘whoami‘}"
# nomeutente1 non è stato dichiarato.
# Viene visualizzato.

nomeutente2=
# nomeutente2 è stato dichiarato, ma contiene un valore nullo.
echo "nomeutente2 = ${nomeutente2:-‘whoami‘}"
# Viene visualizzato perché sono stati utilizzati :- al posto del semplice -.

exit 0
```

Il costrutto *parametro-default* viene utilizzato per fornire agli script gli argomenti "dimenticati" da riga di comando.

```
DEFAULT_NOMEFILE=generico.dat
nomefile=${1:-$DEFAULT_NOMEFILE}
#  Se non diversamente specificato, il successivo blocco di
#+ comandi agisce sul file "generico.dat".
#
#  Seguono comandi.
```

Vedi anche Example 3-4, Example 29-2 e Example A-7.

Si confronti questo metodo per fornire un argomento di default con l'uso di una *lista and*.

`${parametro=default}` `${parametro:=default}`

Se parametro non è impostato, viene impostato al valore fornito da default.

Le due forme sono quasi equivalenti. I : servono solo quando *$parametro* è stato dichiarato, ma non impostato, [4] come visto in precedenza.

```
echo ${nomeutente=‘whoami‘}
# La variabile "nomeutente" è stata ora impostata con ‘whoami‘.
```

${parametro+altro_valore}
${parametro:+altro_valore}

Se parametro è impostato, assume **altro_valore**, altrimenti viene impostato come stringa nulla.

Le due forme sono quasi equivalenti. I : servono solo quando *parametro* è stato dichiarato, ma non impostato. Vedi sopra.

```
echo "###### \${parametro+altro_valore} ########"
echo

a=${param1+xyz}
echo "a = $a"      # a =

param2=
a=${param2+xyz}
echo "a = $a"      # a = xyz

param3=123
a=${param3+xyz}
echo "a = $a"      # a = xyz

echo
echo "###### \${parametro:+altro_valore} ########"
echo

a=${param4:+xyz}
echo "a = $a"      # a =

param5=
a=${param5:+xyz}
echo "a = $a"      # a =
# Risultato diverso da   a=${param5+xyz}

param6=123
a=${param6+xyz}
echo "a = $a"      # a = xyz
```

${parametro?msg_err}
${parametro:?msg_err}

Se parametro è impostato viene usato, altrimenti visualizza un messaggio d'errore (msg_err).

Le due forme sono quasi equivalenti. I : servono solo quando *parametro* è stato dichiarato, ma non impostato. Come sopra.

Example 9-13. Sostituzione di parametro e messaggi d’errore

```
#!/bin/bash

# Verifica alcune delle variabili d’ambiente di sistema.
#  Se, per sempio, $USER, il nome dell’utente corrente, non è impostata,
#+ la macchina non può riconoscervi.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Il nome della macchina è $HOSTNAME."
  echo "Tu sei $USER."
  echo "La directory home è $HOME."
  echo "La cartella di posta INBOX si trova in $MAIL."
  echo
  echo "Se leggete questo messaggio, vuol dire che"
  echo "le variabili d’ambiente più importanti sono impostate."
  echo
  echo

# ------------------------------------------------------

#  Il costrutto ${nomevariabile?} può verificare anche
#+ le variabili impostate in uno script.

QuestaVariabile=Valore-di-Questa-Variabile
#  È da notare, en passant, che le variabili stringa possono contenere
#+ caratteri che non sono consentiti se usati nei loro nomi .
: ${QuestaVariabile?}
echo "Il valore di QuestaVariabile è $QuestaVariabile".
echo
echo


: ${ZZXy23AB?"ZZXy23AB non è stata impostata."}
#  Se ZZXy23AB non è stata impostata,
#+ allora lo script termina con un messaggio d’errore.

# Il messaggio d’errore può essere specificato.
# : ${ZZXy23AB?"ZZXy23AB non è stata impostata."}


#  Stesso risultato con:
#+     finta_variabile=${ZZXy23AB?}
#+     finta_variabile=${ZZXy23AB?"ZXy23AB non è stata impostata."}
#
#      echo ${ZZXy23AB?} >/dev/null



echo "Questo messaggio non viene visualizzato perché lo script è terminato prima."

QUI=0
```

```
exit $QUI   # *Non* termina in questo punto.
```

Example 9-14. Sostituzione di parametro e messaggi “utilizzo”

```
#!/bin/bash
# usage-message.sh

: ${1?"Utilizzo: $0 ARGOMENTO"}
#  Lo script termina qui, se non vi è un parametro da riga di comando,
#+ e viene visualizzato il seguente messaggio d’errore.
#    usage-message.sh: 1: Utilizzo: usage-message.sh ARGOMENTO

echo "Queste due righe vengono visualizzate solo se è stato
fornito un argomento."
echo "argomento da riga di comando = \"$1\""

exit 0  #  Lo script termina a questo punto solo se è stato
        #+ eseguito con l’argomento richiesto.

# Verificate l’exit status dello script eseguito, sia con che senza argomento.
# Se il parametro è stato fornito, allora  "$?" è  0.
# Altrimenti "$?" è 1.
```

Sostituzione ed espansione di parametro. Le espressioni che seguono sono il complemento delle operazioni sulle stringhe del costrutto **match** con **expr** (vedi Example 12-7). Vengono per lo più usate per la verifica dei nomi dei file.

Lunghezza della variabile / rimozione di sottostringa

`${#var}`

`Lunghezza della stringa` (numero dei caratteri di `$var`). Nel caso di un array, **${#array}** rappresenta la lunghezza del primo elemento dell’array.

> **Note:** Eccezioni:
>
> - **${#*}** e **${#@}** forniscono il *numero dei parametri posizionali*.
> - Per gli array, **${#array[*]}** e **${#array[@]}** forniscono il numero degli elementi che compongono l’array.

Example 9-15. Lunghezza di una variabile

```
#!/bin/bash
# length.sh

E_NO_ARG=65

if [ $# -eq 0 ]  # Devono essere forniti degli argomenti allo script.
then
  echo "Eseguite lo script con uno o più argomenti."
```

```
  exit $E_NO_ARG
fi

var01=abcdEFGH28ij

echo "var01 = ${var01}"
echo "Lunghezza di var01 = ${#var01}"

echo "Numero di argomenti passati allo script = ${#@}"
echo "Numero di argomenti passati allo script = ${#*}"

exit 0
```

${var#Modello}
${var##Modello}

Toglie da `$var` la parte più breve/lunga di `$Modello` verificata all'*`inizio`* di `$var`.

Una dimostrazione del suo impiego tratta dall'Example A-8:

```
# Funzione dall'esempio "days-between.sh".
# Toglie lo/gli zeri iniziali dall'argomento fornito.

toglie_zero_iniziale () #  Toglie possibili zeri iniziali
{                       #+ dagli argomenti passati.
  return=${1#0}         #  "1" stà per $1 -- l'argomento passato.
}                       #  "0" indica ciò che va tolto da "$1" -- gli zeri.
```

Variante, più elaborata dell'esempio precedente, di Manfred Schwarb:

```
toglie_zero_iniziale2 () #  Toglie possibili zeri iniziali, altrimenti
{                        #+ Bash interpreta tali numeri come valori ottali.
  shopt -s extglob       #  Abilita il globbing esteso.
  local val=${1##+(0)}   #  Usa una variabile locale, verifica d'occorrenza più
                         #+ lunga delle serie di 0.
  shopt -u extglob       #  Disabilita il globbing esteso.
  _toglie_zero_iniziale2=${val:-0}
                         #  Nel caso l'input sia 0, restituisce 0 invece di "".
}
```

Altro esempio di utilizzo:

```
echo ‘basename $PWD‘        # Nome della directory di lavoro corrente.
echo "${PWD##*/}"           # Nome della directory di lavoro corrente.
echo
echo ‘basename $0‘          # Nome dello script.
echo $0                     # Nome dello script.
echo "${0##*/}"             # Nome dello script.
echo
nomefile=test.dat
echo "${nomefile##*.}"      # dat
```

```
                                # Estensione del nome del file.
```

`${var%Modello}`
`${var%%Modello}`

Toglie da `$var` la parte più breve/lunga di `$Modello` verificata alla *`fine`* di `$var`.

La versione 2 di Bash possiede delle opzioni aggiuntive.

Example 9-16. Ricerca di corrispondenza nella sostituzione di parametro

```
#!/bin/bash
# patt-matching.sh

#  Ricerca di corrispondenza utilizzando gli operatori si sostituzione
#+ di parametro # ##  % %%.

var1=abcd12345abc6789
modello1=a*c  #  * (carattere jolly) verifica tutto quello che
              #+ è compreso tra a - c.

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789
                              # (forma alternativa)
echo "Numero di caratteri in ${var1} = ${#var1}"
echo "modello1 = $modello1"   # a*c  (tutto ciò che è compreso tra 'a' e 'c')
echo


echo '${var1#$modello1}  =' "${var1#$modello1}"    # d12345abc6789
#  All'eventuale occorrenza più corta, toglie i primi 3 caratteri
#+ abcd12345abc6789                               ^^^^^
#  |-|
echo '${var1##$modello1} =' "${var1##$modello1}"   # 6789
#  All'eventuale occorrenza più lunga, toglie i primi 12 caratteri
#+ abcd92345abc6789                               ^^^^^
#+ |----------|

echo; echo

modello2=b*9            # tutto quello che si trova tra 'b' e '9'
echo "var1 = $var1"     # Ancora  abcd12345abc6789
echo "modello2 = $modello2"
echo

echo '${var1%modello2}  =' "${var1%$modello2}"     #     abcd12345a
#  All'eventuale occorrenza più corta, toglie gli ultimi 6 caratteri
#+ abcd12345abc6789                                 ^^^^^^
#+           |----|
echo '${var1%%modello2} =' "${var1%%$modello2}"    #     a
```

```
#  All'eventuale occorrenza più lunga, toglie gli ultimi 15 caratteri
#+ abcd12345abc6789                                ^^^^^^
#+  |-------------|

#  Ricordate, # e ## agiscono sulla parte iniziale della stringa
#+            (da sinistra verso destra), % e %% agiscono sulla parte
#+            finale della stringa (da destra verso sinistra).

echo

exit 0
```

Example 9-17. Rinominare le estensioni dei file:

```
#!/bin/bash

#                ref
#                ---

# Rinomina (le) estensioni (dei) file.
#
#         ref vecchia_estensione nuova_estensione
#
# Esempio:
# Per rinominare tutti i file *.gif della directory di lavoro in *.jpg,
#          ref gif jpg

ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ]
then
  echo "Utilizzo: 'basename $0' vecchia_estensione nuova_estensione"
  exit $E_ERR_ARG
fi

for nomefile in *.$1
# Passa in rassegna l'elenco dei file che terminano con il 1mo argomento.
do
  mv $nomefile ${nomefile%$1}$2
  #  Toglie la parte di nomefile che verifica il 1mo argomento,
  #+ quindi aggiunge il 2do argomento.
done

exit 0
```

Espansione di variabile / Sostituzione di sottostringa

I costrutti seguenti sono stati adottati da *ksh*.

${var:pos}

La variabile *var* viene espansa iniziando da *pos*.

${var:pos:lun}

Espansione di un massimo di *lun* caratteri della variabile *var*, iniziando da *pos*. Vedi Example A-15 per una dimostrazione dell'uso creativo di questo operatore.

${var/Modello/Sostituto}

La prima occorrenza di *Modello* in *var* viene rimpiazzata da *Sostituto*.

Se si omette *Sostituto* allora la prima occorrenza di *Modello* viene rimpiazzata con *niente*, vale a dire, cancellata.

${var//Modello/Sostituto}

Sostituzione globale. Tutte le occorrenze di *Modello* presenti in *var* vengono rimpiazzate da *Sostituto*.

Come prima, se si omette *Sostituto* allora tutte le occorrenze di *Modello* vengono rimpiazzate con *niente*, vale a dire, cancellate.

Example 9-18. Utilizzare la verifica di occorrenza per controllare stringhe arbitrarie

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (viene tolto tutto ciò che si trova prima del primo"
echo "trattino, compreso) = $t"
#  t=${var1#*-}  Dà lo stesso risultato,
#+ perché # verifica la stringa più corta,
#+ e * verifica tutto quello che sta prima, compresa una stringa vuota.
# (Grazie, S. C. per la puntualizzazione.)

t=${var1##*-*}
echo "Se var1 contiene un \"-\", viene restituita una stringa vuota..."
echo "var1 = $t"


t=${var1%*-*}
echo "var1 (viene tolto tutto ciò che si trova dopo l'ultimo"
echo "trattino, compreso) = $t"

echo

# ------------------------------------------
percorso=/home/bozo/idee/pensieri.di.oggi
# ------------------------------------------
echo "percorso = $percorso"
t=${percorso##/*/}
```

```
echo "percorso senza tutti i prefissi = $t"
#  Stesso risultato con  t=‘basename $percorso‘ , in questo caso particolare.
#  t=${percorso%/}; t=${t##*/}  è una soluzione più generica,
#+ ma talvolta potrebbe non funzionare.
#  Se $percorso termina con un carattere di ritorno a capo, allora
#+ ‘basename $percorso‘ fallisce, al contrario dell’espressione precedente.
#  (Grazie, S.C.)

t=${percorso%/*.*}
# Stesso risultato di   t=‘dirname $percorso‘
echo "percorso a cui è stato tolto il suffisso (/pensieri.di.oggi)  = $t"
#  Questi operatori possono non funzionare, come nei casi"../",
#+ "/foo////", # "foo/", "/". Togliere i suffissi, specialmente quando
#+ basename non ne ha, ma dirname sì, complica la faccenda.
# (Grazie, S.C.)

echo

t=${percorso:11}
echo "$percorso, senza i primi 11 caratteri = $t"
t=${percorso:11:5}
echo "$percorso, senza i primi 11 caratteri e ridotto alla \
lunghezza di 5 caratteri = $t"

echo

t=${percorso/bozo/clown}
echo "$percorso con \"bozo\" sostituito da \"clown\" = $t"
t=${percorso/oggi/}
echo "$percorso con \"oggi\" cancellato = $t"
t=${percorso//o/O}
echo "$percorso con tutte le o minuscole cambiate in O maiuscole = $t"
t=${percorso//o/}
echo "$percorso da cui sono state cancellate tutte le o = $t"

exit 0
```

${var/#Modello/Sostituto}

Se il *prefisso* di *var* è verificato da *Modello*, allora *Sostituto* rimpiazza *Modello*.

${var/%Modello/Sostituto}

Se il *suffisso* di *var* è verificato da *Modello*, allora *Sostituto* rimpiazza *Modello*.

Example 9-19. Verifica di occorrenza di prefissi o suffissi di stringa

```
#!/bin/bash
# Sostituzione di occorrenza di prefisso/suffisso di stringa.

v0=abc1234zip1234abc    # Variabile originale.
echo "v0 = $v0"         # abc1234zip1234abc
```

```
echo

# Verifica del prefisso (inizio) della stringa.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDEF1234zip1234abc
                        # |----|

# Verifica del suffisso (fine) della stringa.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #               |----|

echo

#  --------------------------------------------------------
#  La verifica deve avvenire all'inizio/fine della stringa,
#+ altrimenti non verrà eseguita alcuna sostituzione.
#  --------------------------------------------------------
v3=${v0/#123/000}       # È verificata, ma non all'inizio.
echo "v3 = $v3"         # abc1234zip1234abc
                        # NESSUNA SOSTITUZIONE.
v4=${v0/%123/000}       # È stata verificata, ma non alla fine.
echo "v4 = $v4"         # abc1234zip1234abc
                        # NESSUNA SOSTITUZIONE.

exit 0
```

${!prefissovar*}
${!prefissovar@}

Verifica tutte le variabili precedentemente dichiarate i cui nomi iniziano con *prefissovar*.

```
xyz23=qualsiasi_cosa
xyz24=

a=${!xyz*}      #  Espande i nomi delle variabili dichiarate che iniziano
                #+ con "xyz".
echo "a = $a"   # a = xyz23 xyz24
a=${!xyz@}      # Come prima.
echo "a = $a"   # a = xyz23 xyz24

# La versione 2.04 di Bash possiede questa funzionalità.
```

9.4. Tipizzare le variabili: declare o typeset

I builtin **declare** o **typeset** (sono sinonimi esatti) consentono di limitare le proprietà delle variabili. È una forma molto debole di tipizzazione, se confrontata con quella disponibile per certi linguaggi di programmazione. Il comando **declare** è specifico della versione 2 o successive di Bash. Il comando **typeset** funziona anche negli script ksh.

opzioni declare/typeset

-r *readonly (sola lettura)*

```
declare -r var1
```

(**`declare -r var1`** è uguale a **`readonly var1`**)

È approssimativamente equivalente al qualificatore di tipo **const** del C. Il tentativo di modificare il valore di una variabile in sola lettura fallisce generando un messaggio d'errore.

-i *intero*

```
declare -i numero
# Lo script tratterà le successive occorrenze di "numero" come un intero.

numero=3
echo "numero = $numero"     # Numero = 3

numero=tre
echo "Numero = $numero"     # numero = 0
# Cerca di valutare la stringa "tre" come se fosse un intero.
```

Sono consentire alcune operazioni aritmetiche sulle variabili dichiarate interi senza la necessità di usare expr o let.

```
n=6/3
echo "n = $n"       # n = 6/3

declare -i n
n=6/3
echo "n = $n"       # n = 2
```

-a *array*

```
declare -a indici
```

La variabile `indici` verrà trattata come un array.

-f *funzioni*

```
declare -f
```

In uno script, una riga con **`declare -f`** senza alcun argomento, elenca tutte le funzioni precedentemente definite in quello script.

```
declare -f nome_funzione
```

Un **`declare -f nome_funzione`** elenca solo la funzione specificata.

-x export

```
declare -x var3
```

Dichiara la variabile come esportabile al di fuori dell'ambiente dello script stesso.

-x var=$valore

```
declare -x var3=373
```

Il comando **declare** consente di assegnare un valore alla variabile mentre viene dichiarata, impostando così anche le sue proprietà.

Example 9-20. Utilizzare declare per tipizzare le variabili

```
#!/bin/bash

funz1 ()
{
echo Questa è una funzione.
}

declare -f        # Elenca la funzione precedente.

echo

declare -i var1   # var1 è un intero.
var1=2367
echo "var1 dichiarata come $var1"
var1=var1+1       #  La dichiarazione di intero elimina la necessità di
                  #+ usare 'let'.
```

```
echo "var1 incrementata di 1 diventa $var1."
# Tentativo di modificare il valore di una variabile dichiarata come intero
echo "Tentativo di modificare var1 nel valore in virgola mobile 2367.1."
var1=2367.1       # Provoca un messaggio d'errore, la variabile non cambia.
echo "var1 è ancora $var1"

echo

declare -r var2=13.36         #  'declare' consente di impostare la proprietà
                              #+ della variabile e contemporaneamente
                              #+ assegnarle un valore.
echo "var2 dichiarata come $var2"
                              #  Tentativo di modificare una variabile in sola
                              #+ lettura.
var2=13.37                    #  Provoca un messaggio d'errore e l'uscita dallo
                              #+ script.

echo "var2 è ancora $var2"    # Questa riga non verrà eseguita.

exit 0                        # Lo script non esce in questo punto.
```

9.5. Referenziazione indiretta di variabili

Ipotizziamo che il valore di una variabile sia il nome di una seconda variabile. È in qualche modo possibile recuperare il valore di questa seconda variabile dalla prima? Per esempio, se `a=lettera_alfabeto` e `lettera_alfabeto=z`, può una referenziazione ad `a` restituire `z`? In effetti questo è possibile e prende il nome di *referenziazione indiretta*. Viene utilizzata l'insolita notazione `eval var1=\$$var2`.

Example 9-21. Referenziazioni indirette

```
#!/bin/bash
# Referenziazione indiretta a variabile.

a=lettera_alfabeto
lettera_alfabeto=z

echo

# Referenziazione diretta.
echo "a = $a"

# Referenziazione indiretta.
eval a=\$$a
echo "Ora a = $a"

echo


# Proviamo a modificare la referenziazione di secondo ordine.
```

```
t=tabella_cella_3
tabella_cella_3=24
echo "\"tabella_cella_3\" = $tabella_cella_3"
echo -n "\"t\" dereferenziata = "; eval echo \$$t
# In questo caso, funziona anche
#   eval t=\$$t; echo "\"t\" = $t"
#  (perché?).

echo

t=tabella_cella_3
NUOVO_VAL=387
tabella_cella_3=$NUOVO_VAL
echo "Valore di \"tabella_cella_3\" modificato in $NUOVO_VAL."
echo "\"tabella_cella_3\" ora $tabella_cella_3"
echo -n "\"t\" dereferenziata  "; eval echo \$$t
# "eval" ha due argomenti "echo" e "\$$t" (impostata a $tabella_cella_3)
echo

# (Grazie, S.C., per aver chiarito il comportamento precedente.)


#  Un altro metodo è quello della notazione ${!t}, trattato nella
#+ sezione "Bash, versione 2". Vedi anche l'esempio "ex78.sh".

exit 0
```

Example 9-22. Passare una referenziazione indiretta a *awk*

```
#!/bin/bash

#  Altra versione dello script "column totaler"
#+ che aggiunge una colonna (contenente numeri) nel file di destinazione.
#  Viene utilizzata la referenziazione indiretta.

ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ] #  Verifica il corretto nr. di argomenti da riga
                     #+ di comando.
then
   echo "Utilizzo: `basename $0` nomefile numero_colonna"
   exit $E_ERR_ARG
fi

nomefile=$1
numero_colonna=$2

#===== Fino a questo punto è uguale all'originale =====#


# Script awk di più di una riga vengono invocati con   awk ' ..... '
```

```
# Inizio script awk.
# ------------------------------------------------------
awk "

{ totale += \$${numero_colonna} # referenziazione indiretta
}
END {
     print totale
     }

     " "$nomefile"
# ---------------------------------------------
# Fine script awk.

#  La referenziazione indiretta evita le difficoltà della referenziazione
#+ di una variabile di shell all'interno di uno script awk incorporato.
#  Grazie, Stephane Chazelas.

exit 0
```

> **Caution**
>
> Questo metodo è un po' complicato. Se la seconda variabile modifica il proprio valore, allora la prima deve essere correttamente dereferenziata (come nell'esempio precedente). Fortunatamente, la notazione *`${!variabile}`*, introdotta con la versione 2 di Bash (vedi Example 35-2), rende la referenziazione indiretta più intuitiva.

9.6. $RANDOM: genera un intero casuale

$RANDOM è una funzione interna di Bash (non una costante) che restituisce un intero *pseudocasuale* nell'intervallo 0 - 32767. $RANDOM *non* dovrebbe essere utilizzata per generare una chiave di cifratura.

Example 9-23. Generare numeri casuali

```
#!/bin/bash

# $RANDOM restituisce un intero casuale diverso ad ogni chiamata.
# Intervallo nominale: 0 - 32767 (intero con segno di 16-bit).

NUM_MASSIMO=10
contatore=1

echo
echo "$NUM_MASSIMO numeri casuali:"
echo "-----------------"
while [ "$contatore" -le $NUM_MASSIMO ]   #  Genera 10 ($NUM_MASSIMO)
                                          #+ interi casuali.
do
```

```
  numero=$RANDOM
  echo $numero
  let "contatore += 1"  # Incrementa il contatore.
done
echo "-----------------"

#  Se è necessario un intero casuale entro un dato intervallo, si usa
#+ l'operatore 'modulo', che restituisce il resto di una divisione.

INTERVALLO=500

echo

numero=$RANDOM
let "numero %= $INTERVALLO"
echo "Il numero casuale è inferiore a $INTERVALLO  ---  $numero"

echo

#  Se è necessario un intero casuale non inferiore a un certo limite,
#+ occorre impostare una verifica per eliminare tutti i numeri al di
#+ sotto di tale limite.

LIMITE_INFERIORE=200

numero=0   # inizializzazione
while [ "$numero" -le $LIMITE_INFERIORE ]
do
  numero=$RANDOM
done
echo "Numero casuale maggiore di $LIMITE_INFERIORE ---  $numero"

echo

#  Le due tecniche precedenti possono essere combinate per ottenere un
#+ numero compreso tra due limiti.

numero=0   # inizializzazione
while [ "$numero" -le $LIMITE_INFERIORE ]
do
  numero=$RANDOM
  let "numero %= $INTERVALLO"  # Riduce $numero entro $INTERVALLO.
done
echo "Numero casuale tra $LIMITE_INFERIORE e $INTERVALLO ---  $numero"
echo


# Genera una scelta binaria, vale a dire, il valore "vero" o "falso".
BINARIO=2
numero=$RANDOM
T=1

let "numero %= $BINARIO"
```

```
# let "numero >>= 14"    dà una migliore distribuzione casuale
# (lo scorrimento a destra elimina tutto tranne l'ultima cifra binaria).
if [ "$numero" -eq $T ]
then
  echo "VERO"
else
  echo "FALSO"
fi

echo


# Si può simulare il lancio dei dadi.
MODULO=7   # Modulo 7 per un intervallo fino a 6.
ZERO=0
dado1=0
dado2=0

#  Si lancia ciascun dado separatamente in modo da ottenere la corretta
#+ probabilità.

  while [ "$dado1" -eq $ZERO ]      # Lo zero non può uscire.
  do
    let "dado1 = $RANDOM % $MODULO" # Lancio del primo dado.
  done

  while [ "$dado2" -eq $ZERO ]
  do
    let "dado2 = $RANDOM % $MODULO" # Lancio del secondo dado.
  done

let "punteggio = $dado1 + $dado2"
echo "Lancio dei dadi = $punteggio"
echo



exit 0
```

Example 9-24. Scegliere una carta a caso dal mazzo

```
#!/bin/bash
# pick-card.sh
# Esempio di scelta a caso di un elemento di un array.

# Scegliete una carta, una carta qualsiasi.

Semi="Fiori
Quadri
Cuori
Picche"
```

```
Denominazioni="2
3
4
5
6
7
8
9
10
Fante
Donna
Re
Asso"


seme=($Semi)                # Inizializza l'array.
denominazione=($Denominazioni)

num_semi=${#seme[*]}        # Conta gli elementi dell'array.
num_denominazioni=${#denominazione[*]}

echo -n "${denominazione[$((RANDOM%num_denominazioni))]} di "
echo ${seme[$((RANDOM%num_semi))]}


# $bozo sh pick-cards.sh
# Fante di Fiori


# Grazie, "jipe," per aver puntualizzato quest'uso di $RANDOM.
exit 0
```

Jipe ha evidenziato una serie di tecniche per generare numeri casuali in un intervallo dato.

```
#  Generare un numero casuale compreso tra 6 e 30.
cnumero=$((RANDOM%25+6))

#  Generare un numero casuale, sempre nell'intervallo 6 - 30,
#+ ma che deve essere divisibile per 3.
cnumero=$(((RANDOM%30/3+1)*3))

#  È da notare che questo non sempre funziona.
#  Fallisce quando $RANDOM restituisce 0.

#  Esercizio: Cercate di capire il funzionamento di questo esempio.
```

Bill Gradwohl ha elaborato una formula più perfezionata che funziona con i numeri positivi.

```
cnumero=$(((RANDOM%(max-min+divisibilePer))/divisibilePer*divisibilePer+min))
```

Qui Bill presenta una versatile funzione che restituisce un numero casuale compreso tra due valori specificati.

Example 9-25. Numero casuale in un intervallo dato

```
#!/bin/bash
# random-between.sh
#  Numero casuale compreso tra due valori specificati.
#  Script di Bill Gradwohl, con modifiche di secondaria importanza fatte
#+ dall'autore del libro.
#  Utilizzato con il permesso dell'autore.


interCasuale() {
   #  Genera un numero casuale positivo o negativo
   #+ compreso tra $min e $max
   #+ e divisibile per $divisibilePer.
   #  Restituisce una distribuzione di valori "ragionevolmente casuale".
   #
   #  Bill Gradwohl - 1 Ott, 2003

   sintassi() {
   # Funzione inserita in un'altra.
      echo
      echo    "Sintassi: interCasuale [min] [max] [multiplo]"
      echo
      echo    "Si aspetta che vengano passati fino a 3 parametri,"
      echo    "tutti però opzionali."
      echo    "min è il valore minimo"
      echo    "max è il valore massimo"
      echo    "multiplo specifica che il numero generato deve essere un"
      echo    "multiplo di questo valore."
      echo    "    cioè divisibile esattamente per questo numero."
      echo
      echo    "Se si omette qualche valore, vengono usati"
      echo    "quelli preimpostati: 0 32767 1"
      echo    "L'esecuzione senza errori restituisce 0, altrimenti viene"
      echo    "richiamata la funzione sintassi e restituito 1."
      echo    "Il numero generato viene restituito nella variabile globale"
      echo    "interCasualeNum"
      echo    "Valori negativi passati come parametri vengono gestiti"
      echo    "correttamente."
   }

   local min=${1:-0}
   local max=${2:-32767}
   local divisibilePer=${3:-1}
   #  Assegnazione dei valori preimpostati, nel caso di mancato passaggio
   #+ dei parametri alla funzione.

   local x
   local intervallo

   # Verifica che il valore di divisibilePer sia positivo.
```

```
[ ${divisibilePer} -lt 0 ] && divisibilePer=$((0-divisibilePer))

# Controllo di sicurezza.
if [ $# -gt 3 -o ${divisibilePer} -eq 0 -o  ${min} -eq ${max} ]; then
   sintassi
   return 1
fi

# Verifica se min e max sono scambiati.
if [ ${min} -gt ${max} ]; then
   # Li scambia.
   x=${min}
   min=${max}
   max=${x}
fi

#  Se min non è esattamente divisibile per $divisibilePer,
#+ viene ricalcolato.
if [ $((min/divisibilePer*divisibilePer)) -ne ${min} ]; then
   if [ ${min} -lt 0 ]; then
      min=$((min/divisibilePer*divisibilePer))
   else
      min=$((((min/divisibilePer)+1)*divisibilePer))
   fi
fi

#  Se max non è esattamente divisibile per $divisibilePer,
#+ viene ricalcolato.
if [ $((max/divisibilePer*divisibilePer)) -ne ${max} ]; then
   if [ ${max} -lt 0 ]; then
      max=$((((max/divisibilePer)-1)*divisibilePer))
   else
      max=$((max/divisibilePer*divisibilePer))
   fi
fi

#  ----------------------------------------------------------------------
#  Ora il lavoro vero.

#  E' da notare che per ottenere una corretta distribuzione dei valori
#+ estremi, si deve agire su un intervallo che va da 0 a
#+ abs(max-min)+divisibilePer, non semplicemente abs(max-min)+1.

#  Il leggero incremento produrrà la giusta distribuzione per i
#+ valori limite.

#  Se si cambia la formula e si usa abs(max-min)+1 si otterranno ancora
#+ dei risultati corretti, ma la loro casualità sarà falsata
#+ dal fatto che il numero di volte in cui verranno restituiti gli estremi
#+ ($min e $max) sarà considerevolmente inferiore a quella ottenuta
#+ usando la formula corretta.
#  ----------------------------------------------------------------------
```

```
   intervallo=$((max-min))
   [ ${intervallo} -lt 0 ] && intervallo=$((0-intervallo))
   let intervallo+=divisibilePer
   interCasualeNum=$(((RANDOM%intervallo)/divisibilePer*divisibilePer+min))

   return 0
}

# Verifica della funzione.
min=-14
max=20
divisibilePer=3


#  Genera un array e controlla che si sia ottenuto almeno uno dei risultati
#+ possibili, se si effettua un numero sufficiente di tentativi.

declare -a risultati
minimo=${min}
massimo=${max}
   if [ $((minimo/divisibilePer*divisibilePer)) -ne ${minimo} ]; then
      if [ ${minimo} -lt 0 ]; then
         minimo=$((minimo/divisibilePer*divisibilePer))
      else
         minimo=$((((minimo/divisibilePer)+1)*divisibilePer))
      fi
   fi


   #  Se max non è esattamente divisibile per $divisibilePer,
   #+ viene ricalcolato.

   if [ $((massimo/divisibilePer*divisibilePer)) -ne ${massimo} ]; then
      if [ ${massimo} -lt 0 ]; then
         massimo=$((((massimo/divisibilePer)-1)*divisibilePer))
      else
         massimo=$((massimo/divisibilePer*divisibilePer))
      fi
   fi


#  Poiché gli indici degli array possono avere solo valori positivi,
#+ è necessario uno spiazzamento che garantisca il raggiungimento
#+ di questo risultato.

spiazzamento=$((0-minimo))
for ((i=${minimo}; i<=${massimo}; i+=divisibilePer)); do
   risultati[i+spiazzamento]=0
done


# Ora si esegue per un elevato numero di volte, per vedere cosa si ottiene.
nr_volte=1000  #  L’autore dello script suggeriva 100000,
```

```
               #+ ma sarebbe occorso veramente molto tempo.

for ((i=0; i<${nr_volte}; ++i)); do

   #  Notate che qui min e max sono specificate in ordine inverso
   #+ per vedere, in questo caso, il corretto comportamento della funzione.

   interCasuale ${max} ${min} ${divisibilePer}

   # Riporta un errore se si verifica un risultato inatteso.
   [ ${interCasualeNum} -lt ${min} -o ${interCasualeNum} -gt ${max} ] \
&& echo errore MIN o MAX  - ${interCasualeNum}!
   [ $((interCasualeNum%${divisibilePer})) -ne 0 ] \
&& echo DIVISIBILE PER errore - ${interCasualeNum}!

   # Registra i risultati statisticamente.
   risultati[interCasualeNum+spiazzamento]=\
$((risultati[interCasualeNum+spiazzamento]+1))
done



# Controllo dei risultati

for ((i=${minimo}; i<=${massimo}; i+=divisibilePer)); do
   [ ${risultati[i+spiazzamento]} -eq 0 ] && echo "Nessun risultato per $i." \
|| echo "${i} generato ${risultati[i+spiazzamento]} volte."
done


exit 0
```

Ma, quant'è casuale $RANDOM? Il modo migliore per verificarlo è scrivere uno script che mostri la distribuzione dei numeri "casuali" generati da $RANDOM. Si lancia diverse volte un dado e si registra ogni volta il risultato...

Example 9-26. Lanciare un dado con RANDOM

```
#!/bin/bash
# Quant'è casuale RANDOM?

RANDOM=$$        #  Cambia il seme del generatore di numeri
                 #+ casuali usando l'ID di processo dello script.

FACCE=6          # Un dado ha 6 facce.
NUMMAX_LANCI=600 # Aumentatelo, se non avete nient'altro di meglio da fare.
lanci=0          # Contatore dei lanci.

tot_zero=0       #  I contatori devono essere inizializzati a 0
tot_uno=0        #+ perché una variabile non inizializzata ha valore nullo,
tot_due=0        #+ non zero.
tot_tre=0
tot_quattro=0
tot_cinque=0
```

```
tot_sei=0

visualizza_risultati ()
{
echo
echo "totale degli uno = $tot_uno"
echo "totale dei due = $tot_due"
echo "totale dei tre = $tot_tre"
echo "totale dei quattro = $tot_quattro"
echo "totale dei cinque = $tot_cinque"
echo "totale dei sei = $tot_sei"
echo
}

aggiorna_contatori()
{
case "$1" in
  0) let "tot_uno += 1";;   #  Poiché un dado non ha lo "zero",
                            #+ lo facciamo corrispondere a 1.
  1) let "tot_due += 1";;   # 1 a 2, ecc.
  2) let "tot_tre += 1";;
  3) let "tot_quattro += 1";;
  4) let "tot_cinque += 1";;
  5) let "tot_sei += 1";;
esac
}

echo


while [ "$lanci" -lt "$NUMMAX_LANCI" ]
do
  let "dado1 = RANDOM % $FACCE"
  aggiorna_contatori $dado1
  let "lanci += 1"
done

visualizza_risultati

#  I punteggi dovrebbero essere distribuiti abbastanza equamente, nell'ipotesi
#+ che RANDOM sia veramente casuale.
#  Con $NUMMAX_LANCI impostata a 600, la frequenza di ognuno dei sei numeri
#+ dovrebbe aggirarsi attorno a 100, più o meno.
#
#  Ricordate che RANDOM è un generatore pseudocasuale, e neanche
#+ particolarmente valido.

# Esercizio (facile):
# ---------------
# Riscrivete lo script per simulare il lancio di una moneta 1000 volte.
# Le possibilità sono "TESTA" o "CROCE".

exit 0
```

Come si è visto nell'ultimo esempio, è meglio "ricalcolare il seme" del generatore `RANDOM` ogni volta che viene invocato. Utilizzando lo stesso seme, `RANDOM` ripete le stesse serie di numeri. (Rispecchiando il comportamento della funzione *`random()`* del C.)

Example 9-27. Cambiare il seme di RANDOM

```
#!/bin/bash
# seeding-random.sh: Cambiare il seme della variabile RANDOM.

MAX_NUMERI=25     # Quantità di numeri che devono essere generati.

numeri_casuali ()
{
contatore=0
while [ "$contatore" -lt "$MAX_NUMERI" ]
do
  numero=$RANDOM
  echo -n "$numero "
  let "contatore += 1"
done
}

echo; echo

RANDOM=1          # Impostazione del seme di RANDOM.
numeri_casuali

echo; echo

RANDOM=1          # Stesso seme...
numeri_casuali    # ...riproduce esattamente la serie precedente.
                  #
                  # Ma, quant'è utile duplicare una serie di numeri "casuali"?

echo; echo

RANDOM=2          # Altro tentativo, ma con seme diverso...
numeri_casuali    # viene generata una serie differente.

echo; echo

# RANDOM=$$  imposta il seme di RANDOM all'id di processo dello script.
# È anche possibile usare come seme di RANDOM i comandi 'time' o 'date'.

# Ancora più elegante...
SEME=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
#  Output pseudocasuale prelevato da /dev/urandom (file di
#+ dispositivo di sistema pseudo-casuale), quindi convertito
#+ con "od" in una riga di numeri (ottali) visualizzabili,
#+ infine "awk" ne recupera solamente uno per SEME.

RANDOM=$SEME
numeri_casuali
```

```
echo; echo

exit 0
```

> **Note:** Il file di dispositivo `/dev/urandom` fornisce un mezzo per generare numeri pseudocasuali molto più "casuali" che non la variabile `$RANDOM`. **`dd if=/dev/urandom of=nomefile bs=1 count=XX`** crea un file di numeri casuali ben distribuiti . Tuttavia, per assegnarli ad una variabile in uno script è necessario un espediente, come filtrarli attraverso od (come nell'esempio precedente) o utilizzare dd (vedi Example 12-43).
>
> Ci sono altri metodi per generare numeri pseudocasuali in uno script. **Awk** ne fornisce uno molto comodo.
>
> **Example 9-28. Numeri pseudocasuali utilizzando awk**
>
> ```
> #!/bin/bash
> # random2.sh: Restituisce un numero pseudo-casuale nell'intervallo 0 - 1.
> # Uso della funzione awk rand().
>
> SCRIPTAWK=' { srand(); print rand() } '
> #            Comando(i) / parametri passati ad awk
> # Notate che srand() ricalcola il seme del generatore di numeri di awk.
>
>
> echo -n "Numeri casuali tra 0 e 1 = "
>
> echo | awk "$SCRIPTAWK"
> # Cosa succede se si omette 'echo'?
>
> exit 0
>
>
> # Esercizi:
> # ---------
>
> # 1) Usando un ciclo, visualizzare 10 differenti numeri casuali.
> #    (Suggerimento: bisogna ricalcolare un diverso seme per la funzione
> #+   "srand()" ad ogni passo del ciclo. Cosa succede se non viene fatto?)
>
> # 2) Usando un intero come fattore di scala, generare numeri casuali
> #+   nell'intervallo tra 10 e 100.
>
> # 3) Come il precedente esercizio nr.2, ma senza intervallo.
> ```
>
> Anche il comando date si presta a generare sequenze di interi pseudocasuali.

9.7. Il costrutto doppie parentesi

Simile al comando let, il costrutto **((...))** consente l'espansione e la valutazione aritmetica. Nella sua forma più semplice, **`a=$(( 5 + 3 ))`** imposta "a" al valore "5 + 3", cioè 8. Non solo, ma questo costrutto consente di gestire, in Bash, le variabili con lo stile del linguaggio C.

Example 9-29. Gestire le variabili in stile C

```
#!/bin/bash
# Manipolare una variabile, in stile C, usando il costrutto ((...)).


echo

(( a = 23 ))  #  Impostazione, in stile C, con gli spazi da entrambi i lati
              #+ dell' "=".
echo "a (valore iniziale) = $a"

(( a++ ))     # Post-incremento di 'a', stile C.
echo "a (dopo a++) = $a"

(( a-- ))     # Post-decremento di 'a', stile C.
echo "a (dopo a--) = $a"

(( ++a ))     # Pre-incremento di 'a', stile C.
echo "a (dopo ++a) = $a"

(( --a ))     # Pre-decremento di 'a', stile C.
echo "a (dopo --a) = $a"

echo

(( t = a<45?7:11 ))   # Operatore ternario del C.
echo "Se a < 45, allora t = 7, altrimenti t = 11."
echo "t = $t "        # Si!

echo


# ---------------------
# Attenzione, sorpresa!
# ---------------------
#  Evidentemente Chet Ramey ha contrabbandato un mucchio di costrutti in
#+ stile C, non documentati, in Bash (in realtà adattati da ksh, in
#+ quantità notevole).
#  Nella documentazione Bash, Ramey chiama ((...)) matematica di shell,
#+ ma ciò va ben oltre l'aritmetica.
#  Mi spiace, Chet, ora il segreto è svelato.

# Vedi anche l'uso del costrutto ((...)) nei cicli "for" e "while".

# Questo costrutto funziona solo nella versione  2.04 e successive, di Bash.

exit 0
```

Vedi anche Example 10-12.

Notes

1. Naturalmente, il PID dello script in esecuzione è $$.
2. I termini “argomento” e “parametro” vengono spesso usati per indicare la stessa cosa. In questo libro hanno lo stesso, identico significato: quello di una variabile passata ad uno script o ad una funzione.
3. Questo vale sia per gli argomenti da riga di comando che per i parametri passati ad una funzione.
4. Se $parametro è nullo, in uno script non interattivo, questo viene terminato con exit status 127 (il codice di errore Bash di “command not found”).

Chapter 10. Cicli ed alternative

Le operazioni sui blocchi di codice sono il cuore di script di shell ben strutturati e organizzati. I costrutti per gestire i cicli e le scelte sono gli strumenti per raggiungere questo risultato.

10.1. Cicli

Un *ciclo* è un blocco di codice che itera (ripete) un certo numero di comandi finché la condizione di controllo del ciclo è vera.

cicli for

for (in)

È il costrutto di ciclo fondamentale. Differisce significativamente dal suo analogo del linguaggio C.

for *arg* in [*lista*]
do
 comando(i)...
done

Note: Ad ogni passo del ciclo, *arg* assume il valore di ognuna delle successive variabili elencate in *lista*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# Al 1° passo del ciclo, $arg = $var1
# Al 2° passo del ciclo, $arg = $var2
# Al 3° passo del ciclo, $arg = $var3
# ...
# Al passo N° del ciclo, $arg = $varN

#  Bisogna applicare il "quoting" agli argomenti della [lista] per
#+ evitare una possibile suddivisione delle parole.
```

Gli argomenti elencati in *lista* possono contenere i caratteri jolly.

Se **do** si trova sulla stessa riga di **for**, è necessario usare il punto e virgola dopo lista.

for *arg* in [*lista*]; do

Example 10-1. Semplici cicli for

```
#!/bin/bash
# Elenco di pianeti.

for pianeta in Mercurio Venere Terra Marte Giove Saturno Urano Nettuno Plutone

do
  echo $pianeta
done

echo

#  L’intera "lista" racchiusa tra apici doppi crea un’unica variabile.

for pianeta in "Mercurio Venere Terra Marte Giove Saturno Urano Nettuno Plutone"
do
  echo $pianeta
done

exit 0
```

> **Note:** Ogni elemento in **`[lista]`** può contenere più parametri. Ciò torna utile quando questi devono essere elaborati in gruppi. In tali casi, si deve usare il comando **set** (vedi Example 11-14) per forzare la verifica di ciascun elemento in **`[lista]`** e per assegnare ad ogni componente i rispettivi parametri posizionali.

Example 10-2. Ciclo for con due parametri in ogni elemento [lista]

```
#!/bin/bash
# Pianeti rivisitati.

# Associa il nome di ogni pianeta con la sua distanza dal sole.

for pianeta in "Mercurio 36" "Venere 67" "Terra 93"  "Marte 142" "Giove 483"
do
  set -- $pianeta  #  Verifica la variabile "pianeta" e imposta i parametri
                   #+ posizionali.
  #  i "--" evitano sgradevoli sorprese nel caso $pianeta sia nulla
  #+ o inizi con un trattino.

  #  Potrebbe essere necessario salvare i parametri posizionali
  #+ originari, perché vengono sovrascritti.
  #  Un modo per farlo è usare un array,
  #        param_origin=("$@")

  echo "$1 $2,000,000 miglia dal sole"

  ##-------due  tab---- servono a concatenare gli zeri al parametro $2
done
```

```
# (Grazie, S.C., per i chiarimenti aggiuntivi.)

exit 0
```

In un ciclo **for**, una variabile può fornire l'elenco **`[lista]`**.

Example 10-3. *Fileinfo:* operare su un elenco di file contenuto in una variabile

```
#!/bin/bash
# fileinfo.sh

FILE="/usr/sbin/privatepw
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind"     # Elenco dei file sui quali volete informazioni.
                  # Compreso il falso file /usr/bin/fakefile.

echo

for file in $FILE
do

  if [ ! -e "$file" ]       # Verifica se il file esiste.
  then
    echo "$file non esiste."; echo
    continue                # Verifica il successivo.
   fi

  ls -l $file | awk '{ print $9 "         dimensione file: " $5 }'
  # Visualizza 2 campi.

  whatis `basename $file`   # Informazioni sul file.
  echo
done

exit 0
```

In un ciclo **for**, **`[lista]`** accetta anche il globbing dei nomi dei file, vale a dire l'uso dei caratteri jolly per l'espansione dei nomi.

Example 10-4. Agire sui file con un ciclo for

```
#!/bin/bash
# list-glob.sh: Generare [lista] in un ciclo for usando il "globbing".

echo

for file in *
do
  ls -l "$file"  # Elenca tutti i file in $PWD (directory corrente).
```

```
  #  Ricordate che il carattere jolly  "*" verifica tutti i file,
  #+ tuttavia, il "globbing" non verifica i file i cui nomi iniziano
  #+ con un punto.

  #  Se il modello non verifica nessun file, allora si autoespande.
  #  Per evitarlo impostate l'opzione nullglob (shopt -s nullglob).
  # Grazie, S.C.
done

echo; echo

for file in [jx]*
do
  rm -f $file    #  Cancella solo i file i cui nomi iniziano con
    #+ "j" o "x" presenti in $PWD.
  echo "Rimosso il file \"$file\"".
done

echo

exit 0
```

Omettere **`in [lista]`** in un ciclo **for** fa sì che il ciclo agisca su $@, l'elenco degli argomenti forniti allo script da riga di comando. Una dimostrazione particolarmente intelligente di ciò è illustrata nell'Example A-17.

Example 10-5. Tralasciare `in [lista]` in un ciclo for

```
#!/bin/bash

# Invocate lo script sia con che senza argomenti e osservate cosa succede.

for a
do
 echo -n "$a "
done

#  Manca 'in lista', quindi il ciclo opera su '$@'
#+ (elenco degli argomenti da riga di comando, compresi gli spazi).

echo

exit 0
```

È possibile impiegare la sostituzione di comando per generare **`[lista]`**. Vedi anche Example 12-40, Example 10-10 ed Example 12-34.

Example 10-6. Generare [lista] in un ciclo for con la sostituzione di comando

```
#!/bin/bash
# Un ciclo for con [lista] prodotta dalla sostituzione di comando.

NUMERI="9 7 3 8 37.53"
```

```
for numero in ‘echo $NUMERI‘  # for numero in 9 7 3 8 37.53
do
  echo -n "$numero "
done

echo
exit 0
```

Ecco un esempio un po’ più complesso dell’utilizzo della sostituzione di comando per creare [lista].

Example 10-7. Un’alternativa con grep per i file binari

```
#!/bin/bash
# bin-grep.sh:  Localizza le stringhe in un file binario.

# Un’alternativa con "grep" per file binari.
# Effetto simile a "grep -a"

E_ERR_ARG=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Utilizzo: ‘basename $0‘ stringa nomefile"
  exit $E_ERR_ARG
fi

if [ ! -f "$2" ]
then
  echo "Il file \"$2\" non esiste."
  exit $E_NOFILE
fi


for parola in $( strings "$2" | grep "$1" )
# Il comando "strings" elenca le stringhe nei file binari.
# L’output viene collegato (pipe) a "grep" che verifica la stringa cercata.
do
  echo $parola
done

#  Come ha sottolineato S.C., il ciclo precedente potrebbe essere
#+ sostituito con la più semplice
#    strings "$2" | grep "$1" | tr -s "$IFS" ’[\n*]’


#  Provate qualcosa come  "./bin-grep.sh mem /bin/ls"  per esercitarvi
#+ con questo script.

exit 0
```

Sempre sullo stesso tema.

Example 10-8. Elencare tutti gli utenti del sistema

```
#!/bin/bash
# userlist.sh

FILE_PASSWORD=/etc/passwd
n=1           # Numero utente

for nome in $(awk 'BEGIN{FS=":"}{print $1}' < "$FILE_PASSWORD" )
# Separatore di campo = :^^^^^^
# Visualizza il primo campo      ^^^^^^^^
# Ottiene l'input dal file delle password    ^^^^^^^^^^^^^^^^^
do
  echo "UTENTE nr.$n = $nome"
  let "n += 1"
done


# UTENTE nr.1 = root
# UTENTE nr.2 = bin
# UTENTE nr.3 = daemon
# ...
# UTENTE nr.30 = bozo

exit 0
```

Esempio finale di [lista] risultante dalla sostituzione di comando.

Example 10-9. Verificare tutti i file binari di una directory in cerca degli autori

```
#!/bin/bash
# findstring.sh:
# Cerca una stringa particolare nei binari di una directory specificata.

directory=/usr/bin/
stringa="Free Software Foundation"  # Vede quali file sono della FSF.

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$stringa" | sed -e "s%$directory%%"
  #  Nell'espressione  "sed", è necessario sostituire il normale
  #+ delimitatore "/" perché si dà il caso che "/" sia uno dei
  #+ caratteri che deve essere filtrato.

done

exit 0

#  Esercizio (facile):
#  -------------------
#  Modificate lo script in modo tale che accetti come parametri da
#+ riga di comando $directory e $stringa.
```

L’output di un ciclo **for** può essere collegato con una pipe ad un comando o ad una serie di comandi.

Example 10-10. Elencare i link simbolici presenti una directory

```
#!/bin/bash
# symlinks.sh: Elenca i link simbolici presenti in una directory.


directory=${1-‘pwd‘}
#  Imposta come predefinita la directory di lavoro corrente, nel caso non ne
#+ venga specificata alcuna.
#  Corrisponde al seguente blocco di codice.
# ------------------------------------------------------------------
# ARG=1                # Si aspetta un argomento da riga di comando.
#
# if [ $# -ne "$ARG" ]  # Se non c’è 1 argomento...
# then
#   directory=‘pwd‘     # directory di lavoro corrente
# else
#   directory=$1
# fi
# ------------------------------------------------------------------

echo "Link simbolici nella directory \"$directory\""

for file in "$( find $directory -type l )"  # -type l = link simbolici
do
  echo "$file"
done | sort                                 #  Se manca sort, l’elenco
    #+ non verrà ordinato.

#  Come ha evidenziato Dominik ’Aeneas’ Schnitzer,
#+ se non si usa il "quoting" per $( find $directory -type l ) i nomi dei
#+ file contenenti spazi non vengono visualizzati correttamente.
#  Il nome viene troncato al primo spazio incontrato.


exit 0

# Jean Helou propone la seguente alternativa:

echo "Link simbolici nella directory \"$directory\""
# Salva l’IFS corrente. Non si è mai troppo prudenti.
VECCHIOIFS=$IFS
IFS=:

for file in $(find $directory -type l -printf "%p$IFS")
do     #                             ^^^^^^^^^^^^^^^^
       echo "$file"
       done|sort
```

Lo `stdout` di un ciclo può essere rediretto in un file, come dimostra la piccola modifica apportata all’esempio precedente.

Example 10-11. Link simbolici presenti in una directory salvati in un file

```
#!/bin/bash
# symlinks.sh: Elenca i link simbolici presenti in una directory.

OUTFILE=symlinks.list                          # file di memorizzazione

directory=${1-‘pwd‘}
#  Imposta come predefinita la directory di lavoro corrente, nel caso non
#+ ne venga specificata alcuna.


echo "Link simbolici nella directory \"$directory\"" > "$OUTFILE"
echo "----------------------------" >> "$OUTFILE"

for file in "$( find $directory -type l )"    # -type l = link simbolici
do
  echo "$file"
done | sort >> "$OUTFILE"                     #  stdout del ciclo rediretto
#           ^^^^^^^^^^^^^                     #+  al file di memorizzazione.

exit 0
```

Vi è una sintassi alternativa per il ciclo **for** che risulta molto familiare ai programmatori in linguaggio C. Si basa sull’uso del costrutto doppie parentesi.

Example 10-12. Un ciclo for in stile C

```
#!/bin/bash
# Due modi per contare fino a 10.

echo

# Sintassi standard.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

#
+=====================================================================+

# Ora facciamo la stessa cosa usando la sintassi in stile C.

LIMITE=10

for ((a=1; a <= LIMITE; a++))  # Doppie parentesi, e "LIMITE" senza "$".
do
  echo -n "$a "
done                           # Un costrutto preso in prestito da ’ksh93’.
```

```
echo; echo

# +==========================================================================+

#  Uso dell' "operatore virgola" del C per incrementare due variabili
#+ contemporaneamente.

for ((a=1, b=1; a <= LIMITE; a++, b++))  # La virgola concatena le operazioni.
do
  echo -n "$a-$b "
done

echo; echo

exit 0
```

Vedi anche Example 26-15, Example 26-16 e Example A-7.

Adesso un *ciclo for* impiegato in un'applicazione "pratica".

Example 10-13. Utilizzare efax in modalità batch

```
#!/bin/bash

ARG_ATTESI=2
E_ERR_ARG=65

if [ $# -ne $ARG_ATTESI ]
# Verifica il corretto numero di argomenti.
then
   echo "Utilizzo: 'basename $0' nr_telefono file_testo"
   exit $E_ERR_ARG
fi


if [ ! -f "$2" ]
then
  echo "Il file $2 non è un file di testo"
  exit $E_ERR_ARG
fi


fax make $2              # Crea file fax formattati dai file di testo.

for file in $(ls $2.0*)  # Concatena i file appena creati.
                         # Usa il carattere jolly in lista.
do
  fil="$fil $file"
done

efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # Esegue il lavoro.
```

```
# Come ha sottolineato S.C. il ciclo for potrebbe essere sostituito con
#    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
# ma non sarebbe stato altrettanto istruttivo [sorriso].

exit 0
```

while

Questo costrutto verifica una condizione data all'inizio del ciclo che viene mantenuto in esecuzione finché quella condizione rimane vera (restituisce exit status 0). A differenza del ciclo for, il *ciclo while* viene usato in quelle situazioni in cui il numero delle iterazioni non è conosciuto in anticipo.

while [*condizione*]
do
 comando...
done

Come nel caso dei cicli for/in, collocare il **do** sulla stessa riga della condizione di verifica rende necessario l'uso del punto e virgola.

while [*condizione*] ; do

È da notare che alcuni cicli **while** specializzati, come per esempio il costrutto getopts, si discostano un po' dalla struttura standard appena illustrata.

Example 10-14. Un semplice ciclo while

```
#!/bin/bash

var0=0
LIMITE=10

while [ "$var0" -lt "$LIMITE" ]
do
  echo -n "$var0 "        # -n sopprime il ritorno a capo.
  var0=‘expr $var0 + 1‘   # var0=$(($var0+1)) anche questa forma va bene.
done

echo

exit 0
```

Example 10-15. Un altro ciclo while

```
#!/bin/bash

echo

while [ "$var1" != "fine" ]     #  while test "$var1" != "fine"
do                              #  altra forma valida.
  echo "Immetti la variabile #1 (fine per terminare) "
  read var1                     #  Non 'read $var1' (perché?).
  echo "variabile #1 = $var1"   #  È necessario il "quoting"
                                #+ per la presenza di "#".
  # Se l'input è 'fine', viene visualizzato a questo punto.
  # La verifica per l'interruzione del ciclo, infatti, è posta all'inizio.

  echo
done

exit 0
```

Un ciclo **while** può avere diverse condizioni. Ma è solamente quella finale che stabilisce quando il ciclo deve terminare. Per questo scopo, però, è necessaria una sintassi leggermente differente.

Example 10-16. Ciclo while con condizioni multiple

```
#!/bin/bash

var1=nonimpostata
precedente=$var1

while echo "variabile-precedente = $precedente"
      echo
      precedente=$var1
      [ "$var1" != fine ] # Tiene traccia del precedente valore di $var1.
      #  "while" con quattro condizioni, ma è solo l'ultima che controlla
      #+ il ciclo.
      #  È l'*ultimo* exit status quello che conta.
do
echo "Immetti la variable nr.1 (fine per terminare) "
  read var1
  echo "variabile nr.1 = $var1"
done

# Cercate di capire come tutto questo funzioni.
# È un tantino complicato.

exit 0
```

Come per il ciclo **for**, anche per un ciclo **while** si può impiegare una sintassi in stile C usando il costrutto doppie parentesi (vedi anche Example 9-29).

Example 10-17. Sintassi in stile C di un ciclo while

```
#!/bin/bash
# wh-loopc.sh: Contare fino a 10 con un ciclo "while".

LIMITE=10
a=1

while [ "$a" -le $LIMITE ]
do
  echo -n "$a "
  let "a+=1"
done           # Fin qui nessuna novità.

echo; echo

# +=================================================================+

# Rifatto con la sintassi del C.

((a = 1))      # a=1
#  Le doppie parentesi consentono gli spazi nell'impostazione di una
#+ variabile, come in C.

while (( a <= LIMITE ))   #  Doppie parentesi senza "$" che precede
                          #+ il nome della variabile.
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
  #  Si.
  #  Le doppie parentesi consentono di incrementare una variabile
  #+ con la sintassi del C.
done

echo

# Ora i programmatori in C si sentiranno a casa loro anche con Bash.

exit 0
```

Note: Un ciclo **while** può avere il proprio `stdin` rediretto da un file tramite un < alla fine del blocco.

until

Questo costrutto verifica una condizione data all'inizio del ciclo che viene mantenuto in esecuzione finché quella condizione rimane falsa (il contrario del ciclo **while**).

until [*condizione-falsa*]
do
  comando...
done

Notate che **until** verifica la condizione all'inizio del ciclo, differendo, in questo, da analoghi costrutti di alcuni linguaggi di programmazione.

Come nel caso dei cicli for/in, collocare il **do** sulla stessa riga della condizione di verifica rende necessario l'uso del punto e virgola.

until [*condizione-falsa*]; do

Example 10-18. Ciclo until

```
#!/bin/bash

until [ "$var1" = fine ] # Condizione di verifica all'inizio del ciclo.
do
  echo "Immetti variabile nr.1 "
  echo "(fine per terminare)"
  read var1
  echo "variabile nr.1 = $var1"
done

exit 0
```

10.2. Cicli annidati

Un ciclo annidato è un ciclo in un ciclo, vale a dire un ciclo posto all'interno del corpo di un altro (chiamato ciclo esterno). Al suo primo passo, il ciclo esterno mette in esecuzione quello interno che esegue il proprio blocco di codice fino alla conclusione. Quindi, al secondo passo, il ciclo esterno rimette in esecuzione quello interno. Questo si ripete finché il ciclo esterno non termina. Naturalmente, un **break** contenuto nel ciclo interno o in quello esterno, può interrompere l'intero processo.

Example 10-19. Cicli annidati

```
#!/bin/bash
# Cicli "for" annidati.

esterno=1             # Imposta il contatore del ciclo esterno.

# Inizio del ciclo esterno.
for a in 1 2 3 4 5
```

```
do
  echo "Passo $esterno del ciclo esterno."
  echo "----------------------"
  interno=1           # Imposta il contatore del ciclo interno.

  # Inizio del ciclo interno.
  for b in 1 2 3 4 5
  do
    echo "Passo $interno del ciclo interno."
    let "interno+=1"  # Incrementa il contatore del ciclo interno.
  done
  # Fine del ciclo interno.

  let "esterno+=1"    #  Incrementa il contatore del ciclo esterno.
  echo                #  Spaziatura tra gli output dei successivi
                      #+ passi del ciclo esterno.
done
# Fine del ciclo esterno.

exit 0
```

Vedi Example 26-11 per un'illustrazione di cicli "while" annidati e Example 26-13 per vedere un ciclo "while" annidato in un ciclo "until".

10.3. Controllo del ciclo

Comandi inerenti al comportamento del ciclo

break
continue

I comandi di controllo del ciclo **break** e **continue** [1] corrispondono esattamente ai loro analoghi negli altri linguaggi di programmazione. Il comando **break** interrompe il ciclo (esce), mentre **continue** provoca il salto all'iterazione successiva, tralasciando tutti i restanti comandi di quel particolare passo del ciclo.

Example 10-20. Effetti di break e continue in un ciclo

```
#!/bin/bash

LIMITE=19  # Limite superiore

echo
echo "Visualizzare i numeri da 1 fino a 20 (saltando 3 e 11)."

a=0

while [ $a -le "$LIMITE" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  #  Esclude  3 e 11
```

```
 then
   continue  #  Salta la parte restante di questa particolare
             #+ iterazione del ciclo.
 fi

 echo -n "$a "
done

# Esercizio:
# Perché il ciclo visualizza fino a 20?

echo; echo

echo Visualizza i numeri da 1 a 20, ma succede qualcosa dopo il 2.

##########################################################################

# Stesso ciclo, ma sostituendo 'continue' con 'break'.

a=0

while [ "$a" -le "$LIMITE" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # Salta l'intero ciclo.
 fi

 echo -n "$a "
done

echo; echo; echo

exit 0
```

Il comando **break** può avere un parametro. Il semplice **break** conclude il ciclo in cui il comando di trova, mentre un **break N** interrompe il ciclo al livello *N*.

Example 10-21. Interrompere un ciclo ad un determinato livello

```
#!/bin/bash
# break-levels.sh: Interruzione di cicli.

# "break N" interrompe i cicli al livello N.

for cicloesterno in 1 2 3 4 5
do
  echo -n "Gruppo $cicloesterno:   "

  for ciclointerno in 1 2 3 4 5
  do
```

```
    echo -n "$ciclointerno "

    if [ "$ciclointerno" -eq 3 ]
    then
      break  # Provate   break 2   per vedere il risultato.
             # ("Interrompe" entrambi i cicli, interno ed esterno).
    fi
  done

  echo
done

echo

exit 0
```

Il comando **continue**, come **break**, può avere un parametro. Un semplice **continue** interrompe l'esecuzione dell'iterazione corrente del ciclo e dà inizio alla successiva. **continue N** salta tutte le restanti iterazioni del ciclo in cui si trova e continua con l'iterazione successiva del ciclo `N` di livello superiore.

Example 10-22. Proseguire ad un livello di ciclo superiore

```
#!/bin/bash
# Il comando "continue N", continua all'Nsimo livello.

for esterno in I II III IV V           # ciclo esterno
do
  echo; echo -n "Gruppo $esterno: "

  for interno in 1 2 3 4 5 6 7 8 9 10  # ciclo interno
  do

    if [ "$interno" -eq 7 ]
    then
      continue 2  #  Continua al ciclo di 2° livello, cioè il
         #+ "ciclo esterno". Modificate la riga precedente
  #+ con un semplice "continue" per vedere il
  #+ consueto comportamento del ciclo.
    fi

    echo -n "$interno "  # 8 9 10 non verranno mai visualizzati.
  done

done

echo; echo

# Esercizio:
# Trovate un valido uso di "continue N" in uno script.

exit 0
```

Example 10-23. Uso di “continue N” in un caso reale

```
# Albert Reiner fornisce un esempio di come usare "continue N":
# -----------------------------------------------------------

#  Supponiamo di avere un numero elevato di job che devono essere
#+ eseguiti, con tutti i dati che devono essere trattati contenuti in un
#+ file, che ha un certo nome ed è contenuto in una data directory.
#+ Ci sono diverse macchine che hanno accesso a questa directory e voglio
#+ distribuire il lavoro su tutte queste macchine. Per far questo,
#+ solitamente, utilizzo nohup con il codice seguente su ogni macchina:

while true
do
  for n in .iso.*
  do
    [ "$n" = ".iso.opts" ] && continue
    beta=${n#.iso.}
    [ -r .Iso.$beta ] && continue
    [ -r .lock.$beta ] && sleep 10 && continue
    lockfile -r0 .lock.$beta || continue
    echo -n "$beta: " ‘date‘
    run-isotherm $beta
    date
    ls -alF .Iso.$beta
    [ -r .Iso.$beta ] && rm -f .lock.$beta
    continue 2
  done
  break
done

#  I dettagli, in particolare sleep N, sono specifici per la mia
#+ applicazione, ma la struttura generale è:

while true
do
  for job in {modello}
  do
    {job già terminati o in esecuzione} && continue
    {marca il job come in esecuzione, lo esegue, lo marca come eseguito}
    continue 2
  done
  break        # Oppure ‘sleep 600’ per evitare la conclusione.
done

#  In questo modo lo script si interromperà solo quando non ci saranno
#+ più job da eseguire (compresi i job che sono stati aggiunti durante
#+ il runtime). Tramite l’uso di appropriati lockfile può essere
#+ eseguito su diverse macchine concorrenti senza duplicazione di
#+ calcoli [che, nel mio caso, occupano un paio d’ore, quindi è
#+ veramente il caso di evitarlo]. Inoltre, poiché la ricerca
#+ ricomincia sempre dall’inizio, è possibile codificare le priorità
#+ nei nomi dei file. Naturalmente, questo si potrebbe fare senza
```

```
#+ ‘continue 2’, ma allora si dovrebbe verificare effettivamente se
#+ alcuni job sono stati eseguiti (in questo caso dovremmo cercare
#+ immediatamente il job successivo) o meno (in quest’altro dovremmo
#+ interrompere o sospendere l’esecuzione per molto tempo prima di
#+ poter verificare un nuovo job).
```

Caution

Il costrutto **continue N** è difficile da capire e complicato da usare, in modo significativo, in qualsiasi contesto. Sarebbe meglio evitarlo.

10.4. Verifiche ed alternative

I costrutti **case** e **select**, tecnicamente parlando, non sono cicli, dal momento che non iterano l’esecuzione di un blocco di codice. Come i cicli, tuttavia, hanno la capacità di dirigere il flusso del programma in base alle condizioni elencate dall’inizio alla fine del blocco.

Controllo del flusso del programma in un blocco di codice

case (in) / esac

Il costrutto **case** è l’equivalente shell di **switch** in C/C++. Permette di dirigere il flusso del programma ad uno dei diversi blocchi di codice, in base alle condizioni di verifica. È una specie di scorciatoia di enunciati if/then/else multipli e uno strumento adatto a creare menu.

case "$*variabile*" in
"$*condizione1*")
comando…
;;
"$*condizione2*")
comando…
;;
esac

Note:

- Il "quoting" delle variabili non è obbligatorio, dal momento che la suddivisione delle parole non ha luogo.
- Ogni riga di verifica termina con una parentesi tonda chiusa).
- Ciascun blocco di istruzioni termina con un *doppio* punto e virgola ;;.
- L’intero blocco **case** termina con **esac** (*case* scritto al contrario).

Example 10-24. Utilizzare case

```
#!/bin/bash

echo; echo "Premi un tasto e poi invio."
read Tasto

case "$Tasto" in
  [a-z]   ) echo "Lettera minuscola";;
  [A-Z]   ) echo "Lettera maiuscola";;
  [0-9]   ) echo "Cifra";;
  *       ) echo "Punteggiatura, spaziatura, o altro";;
esac  #  Sono permessi gli intervalli di caratteri se
      #+ compresi in [parentesi quadre].

# Esercizio:
# --------
#
#  Così com'è, lo script accetta la pressione di un solo tasto, quindi
#+ termina. Modificate lo script in modo che accetti un input continuo,
#+ visualizzi ogni tasto premuto e termini solo quando viene digitata
#+ una "X". Suggerimento: racchiudete tutto in un ciclo "while".

exit 0
```

Example 10-25. Creare menu utilizzando case

```
#!/bin/bash

# Un database di indirizzi non molto elegante

clear # Pulisce lo schermo.

echo "          Elenco Contatti"
echo "          ------ --------"
echo "Scegliete una delle persone seguenti:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read persona

case "$persona" in
# Notate l'uso del "quoting" per la variabile.

  "E" | "e" )
  # Accetta sia una lettera maiuscola che minuscola.
  echo
  echo "Roland Evans"
```

```
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Socio d'affari & vecchio amico"
  ;;
# Attenzione al doppio punto e virgola che termina ogni opzione.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Fidanzata"
  echo "Compleanno: Feb. 11"
  ;;

# Aggiungete in seguito le informazioni per Smith & Zane.

         * )
   #  Opzione predefinita.
   #  Un input vuoto (tasto INVIO) o diverso dalle scelte
   #+ proposte, viene verificato qui.
   echo
   echo "Non ancora inserito nel database."
  ;;

esac

echo

#  Esercizio:
#  --------
#  Modificate lo script in modo che accetti un input continuo,
#+ invece di terminare dopo aver visualizzato un solo indirizzo.

exit 0
```

Un uso particolarmente intelligente di **case** è quello per verificare gli argomenti passati da riga di comando.

```
#!/bin/bash

case "$1" in
"") echo "Utilizzo: ${0##*/} <nomefile>"; exit 65;;
                             # Nessun parametro da riga di comando,
                             # o primo parametro vuoto.
#  Notate che ${0##*/} equivale alla sostituzione di parametro
#+ ${var##modello}. Cioè $0.
```

```
-*) NOMEFILE=./$1;;   #  Se il nome del file passato come argomento
                      #+ ($1) inizia con un trattino, lo sostituisce
                      #+ con ./$1 di modo che i comandi successivi
                      #+ non lo interpretino come un'opzione.

* ) NOMEFILE=$1;;     # Altrimenti, $1.
esac
```

Example 10-26. Usare la sostituzione di comando per creare la variabile di case

```
#!/bin/bash
# Usare la sostituzione di comando per creare la variabile di "case".

case $( arch ) in   # "arch" restituisce l'architettura della macchina.
i386 ) echo "Macchina con processore 80386";;
i486 ) echo "Macchina con processore 80486";;
i586 ) echo "Macchina con processore Pentium";;
i686 ) echo "Macchina con processore Pentium2+";;
*    ) echo "Altro tipo di macchina";;
esac

exit 0
```

Un costrutto **case** può filtrare le stringhe in una ricerca che fa uso del globbing.

Example 10-27. Una semplice ricerca di stringa

```
#!/bin/bash
# match-string.sh: semplice ricerca di stringa

verifica_stringa ()
{
  UGUALE=0
  NONUGUALE=90
  PARAM=2     # La funzione richiede 2 argomenti.
  ERR_PARAM=91

  [ $# -eq $PARAM ] || return $ERR_PARAM

  case "$1" in
  "$2") return $UGUALE;;
  *   ) return $NONUGUALE;;
  esac

}


a=uno
b=due
c=tre
d=due
```

```
verifica_stringa $a     # numero di parametri errato
echo $?                 # 91

verifica_stringa $a $b  # diverse
echo $?                 # 90

verifica_stringa $b $d  # uguali
echo $?                 # 0


exit 0
```

Example 10-28. Verificare un input alfabetico

```
#!/bin/bash
# isalpha.sh: Utilizzare la struttura "case" per filtrare una stringa.

SUCCESSO=0
FALLIMENTO=-1

isalpha ()  #  Verifica se il *primo carattere* della stringa
            #+ di input è una lettera.
{
if [ -z "$1" ]               # Nessun argomento passato?
then
  return $FALLIMENTO
fi

case "$1" in
[a-zA-Z]*) return $SUCCESSO;;  # Inizia con una lettera?
*        ) return $FALLIMENTO;;
esac
}             # Confrontatelo con la funzione "isalpha ()" del C.


isalpha2 ()   # Verifica se l'*intera stringa* è composta da lettere.
{
  [ $# -eq 1 ] || return $FALLIMENTO

  case $1 in
  *[!a-zA-Z]*|"") return $FALLIMENTO;;
               *) return $SUCCESSO;;
  esac
}

isdigit ()    # Verifica se l'*intera stringa* è formata da cifre.
{             # In altre parole, verifica se è una variabile numerica.
  [ $# -eq 1 ] || return $FALLIMENTO

  case $1 in
```

```
  *[!0-9]*|"") return $FALLIMENTO;;
            *) return $SUCCESSO;;
  esac
}


verifica_var ()  # Front-end per isalpha ().
{
if isalpha "$@"
then
  echo "\"$*\" inizia con un carattere alfabetico."
  if isalpha2 "$@"
  then     # Non ha significato se il primo carattere non è alfabetico.
    echo "\"$*\" contiene solo lettere."
  else
    echo "\"$*\" contiene almeno un carattere non alfabetico."
  fi
else
  echo "\"$*\"  non inizia con una lettera."
                # Stessa risposta se non viene passato alcun argomento.
fi

echo

}

verifica_cifra ()# Front-end per isdigit ().
{
if isdigit "$@"
then
  echo "\"$*\" contiene solo cifre [0 - 9]."
else
  echo "\"$*\" contiene almeno un carattere diverso da una cifra."
fi

echo

}

a=23skidoo
b=H3llo
c=-Cosa?
d=Cosa?
e=‘echo $b‘      # Sostituzione di comando.
f=AbcDef
g=27234
h=27a34
i=27.34

verifica_var $a
verifica_var $b
verifica_var $c
verifica_var $d
```

```
verifica_var $e
verifica_var $f
verifica_var     # Non viene passato nessun argomento, cosa succede?
#
verifica_cifra $g
verifica_cifra $h
verifica_cifra $i


exit 0           # Script perfezionato da S.C.

# Esercizio:
# --------
#  Scrivete la funzione 'isfloat ()' che verifichi i numeri in virgola
#+ mobile. Suggerimento: la funzione è uguale a 'isdigit ()', ma con
#+ l'aggiunta della verifica del punto decimale.
```

select

Il costrutto **select**, adottato dalla Shell Korn, è anch'esso uno strumento per creare menu.

select *variabile* [in *lista*]
do
comando...
break
done

Viene visualizzato un prompt all'utente affinché immetta una delle scelte presenti nella variabile lista. Si noti che **select** usa, in modo predefinito, il prompt `PS3` (`#?`). Questo può essere modificato.

Example 10-29. Creare menu utilizzando select

```
#!/bin/bash

PS3='Scegli il tuo ortaggio preferito: '# Imposta la stringa del prompt.

echo

select verdura in "fagioli" "carote" "patate" "cipolle" "rape"
do
  echo
  echo "Il tuo ortaggio preferito sono i/le $verdura."
  echo "Yuck!"
  echo
  break  # Cosa succederebbe se non ci fosse il "break"?
done

exit 0
```

Se viene omesso **`in lista`** allora **select** usa l'elenco degli argomenti passati da riga di comando allo script (`$@`) o alla funzione in cui il costrutto **select** è inserito.

Lo si confronti con il comportamento del costrutto

for *variabile* [in *lista*]

con **`in lista`** omesso.

Example 10-30. Creare menu utilizzando select in una funzione

```
#!/bin/bash

PS3='Scegli il tuo ortaggio preferito: '

echo

scelta_di()
{
select verdura
# [in lista] omesso, quindi 'select' usa gli argomenti passati alla funzione.
do
  echo
  echo "Il tuo ortaggio preferito: $verdura."
  echo "Yuck!"
  echo
  break
done
}

scelta_di fagioli riso carote ravanelli pomodori spinaci
#         $1      $2   $3     $4        $5       $6
#         passati alla funzione scelta_di()

exit 0
```

Vedi anche Example 35-3.

Notes

1. Sono builtin di shell, mentre altri comandi di ciclo, come while e case, sono parole chiave.

Chapter 11. Comandi interni e builtin

Un *builtin* è un **comando** appartenente alla serie degli strumenti Bash, letteralmente *incorporato*. Questo è stato fatto sia per motivi di efficienza -- i builtin eseguono più rapidamente il loro compito di quanto non facciano i comandi esterni, che di solito devono generare un processo separato (forking) -- sia perché particolari builtin necessitano di un accesso diretto alle parti interne della shell.

> Quando un comando, o la stessa shell, svolge un certo compito, dà inizio *(spawn)* ad un nuovo sottoprocesso. Questa azione si chiama *forking*. Il nuovo processo è il "figlio", mentre il processo che l'ha *generato* è il "genitore". Mentre il *processo figlio* sta svolgendo il proprio lavoro, il *processo genitore* resta ancora in esecuzione.
>
> In genere, un *builtin* Bash eseguito in uno script non genera un sottoprocesso. Al contrario, un filtro o un comando di sistema esterno, solitamente, *avvia* un sottoprocesso.

Un builtin può avere un nome identico a quello di un comando di sistema. In questo caso Bash lo reimplementa internamente. Per esempio, il comando Bash **echo** non è uguale a `/bin/echo`, sebbene la loro azione sia quasi identica.

```
#!/bin/bash

echo "Questa riga usa il builtin \"echo\"."
/bin/echo "Questa riga usa il comando di sistema  /bin/echo."
```

Una *parola chiave* è un simbolo, un operatore o una parola *riservata*. Le parole chiave hanno un significato particolare per la shell e, difatti, rappresentano le componenti strutturali della sua sintassi . Ad esempio "for", "while", "do" e "!" sono parole chiave. Come un *builtin*, una parola chiave è una componente interna di Bash, ma a differenza di un builtin, non è di per se stessa un comando, ma parte di una struttura di comandi più ampia. [1]

I/O

echo

visualizza (allo `stdout`) un'espressione o una variabile (vedi Example 4-1).

```
echo Ciao
echo $a
```

echo richiede l'opzione `-e` per visualizzare le sequenze di escape. Vedi Example 5-2.

Normalmente, ogni comando **echo** visualizza una nuova riga. L'opzione `-n` annulla questo comportamento.

> **Note: echo** si può utilizzare per fornire una sequenza di comandi in una pipe.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR contiene la sottostringa \"txt\""
fi
```

Note: Si può utilizzare **echo**, in combinazione con la sostituzione di comando, per impostare una variabile.

a=‘echo "CIAO" | tr A-Z a-z‘

Vedi anche Example 12-16, Example 12-2, Example 12-33 ed Example 12-34.

Si faccia attenzione che **echo ‘comando‘** cancella tutti i ritorni a capo generati dall’output di *comando*.

La variabile $IFS (internal field separator), di norma, ha \n (ritorno a capo) tra i suoi caratteri di spaziatura. Bash, quindi, scinde l’output di *comando* in corrispondenza dei ritorni a capo. Le parti vengono passate come argomenti a **echo**. Di conseguenza **echo** visualizza questi argomenti separati da spazi.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
-rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au


bash$ echo ‘ls -l /usr/share/apps/kjezz/sounds‘
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root root 362 Nov 7 2000 seconds.au
```

Note: Questo comando è un builtin di shell e non è uguale a `/bin/echo`, sebbene la sua azione sia simile.

```
bash$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

printf

Il comando **printf**, visualizzazione formattata, rappresenta un miglioramento di **echo**. È una variante meno potente della funzione di libreria `printf()` del linguaggio C. Anche la sua sintassi è un po’ differente.

printf *stringa di formato... parametro...*

È la versione builtin Bash del comando `/bin/printf` o `/usr/bin/printf`. Per una descrizione dettagliata, si veda la pagina di manuale di **printf** (comando di sistema).

Caution

Le versioni più vecchie di Bash potrebbero non supportare **printf**.

Example 11-1. printf in azione

```
#!/bin/bash
# printf demo

PI=3,14159265358979                             # Vedi nota a fine listato
CostanteDecimale=31373
Messaggio1="Saluti,"
Messaggio2="un abitante della Terra."

echo

printf "Pi con 2 cifre decimali = %1.2f" $PI
echo
printf "Pi con 9 cifre decimali = %1.9f" $PI  #  Esegue anche il corretto
                                              #+ arrotondamento.

printf "\n"                                    #  Esegue un ritorno a capo,
                                              #  equivale a 'echo'.

printf "Costante = \t%d\n" $CostanteDecimale  #  Inserisce un carattere
                                              #+ di tabulazione (\t)

printf "%s %s \n" $Messaggio1 $Messaggio2

echo

# ================================================#
# Simulazione della funzione 'sprintf' del C.
# Impostare una variabile con una stringa di formato.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Pi con 12 cifre decimali = $Pi12"

Msg=`printf "%s %s \n" $Messaggio1 $Messaggio2`
echo $Msg; echo $Msg

# Abbiamo la fortuna di poter ora disporre della funzione 'sprintf'
# come modulo caricabile per Bash. Questo, però, non è portabile.

exit 0

#  N.d.T. Nella versione originale veniva usato il punto come separatore
#+ decimale. Con le impostazioni locali italiane il punto avrebbe
#+ impedito il corretto funzionamento di printf.
```

Un'utile applicazione di **printf** è quella di impaginare i messaggi d'errore

```
E_ERR_DIR=65

var=directory_inesistente
```

```
errore()
{
  printf "$@" >&2
  # Organizza i parametri posizionali passati e li invia allo stderr.
  echo
  exit $E_ERR_DIR
}

cd $var || errore $"Non riesco a cambiare in %s." "$var"

# Grazie, S.C.
```

read

"Legge" il valore di una variabile dallo `stdin`, vale a dire, preleva in modo interattivo l'input dalla tastiera. L'opzione `-a` permette a **read** di assegnare le variabili di un array (vedi Example 26-6).

Example 11-2. Assegnamento di variabile utilizzando read

```
#!/bin/bash

echo -n "Immetti il valore della variabile 'var1': "
# L'opzione -n di echo sopprime il ritorno a capo.

read var1
#  Notate che non vi è nessun '$' davanti a var1, perché la variabile
#+ è in fase di impostazione.

echo "var1 = $var1"


echo

# Un singolo enunciato 'read' può impostare più variabili.
echo -n "Immetti i valori delle variabili 'var2' e 'var3' (separati da \
uno spazio o da tab): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
#  Se si immette un solo valore, le rimanenti variabili restano non
#+ impostate (nulle).

exit 0
```

read, senza una variabile associata, assegna l'input alla variabile dedicata $REPLY.

Example 11-3. Cosa succede quando read non è associato ad una variabile

```
#!/bin/bash

echo

# ------------------------- #
# Primo blocco di codice.
echo -n "Immetti un valore: "
read var
echo "\"var\" = "$var""
# Tutto come ci si aspetta.
# ------------------------- #

echo

echo -n "Immetti un altro valore: "
read           #  Non viene fornita alcuna variabile a 'read',
       #+ quindi... l'input di 'read' viene assegnato alla
       #+ variabile predefinita $REPLY.

var="$REPLY"
echo "\"var\" = "$var""
# Stesso risultato del primo blocco di codice.

echo

exit 0
```

Normalmente, immettendo una **** nell'input di **read** si disabilita il ritorno a capo. L'opzione `-r` consente di interpretare la **** letteralmente.

Example 11-4. Input su più righe per read

```
#!/bin/bash

echo

echo "Immettete una stringa che termina con \\, quindi premete <INVIO>."
echo "Dopo di che, immettete una seconda stringa e premete ancora <INVIO>."
read var1     # La "\" sopprime il ritorno a capo durante la lettura di "var1".
              #     prima riga \
              #     seconda riga

echo "var1 = $var1"
#     var1 = prima riga seconda riga

#  Per ciascuna riga che termina con "\", si ottiene un prompt alla riga
#+ successiva per continuare ad inserire caratteri in var1.

echo; echo

echo "Immettete un'altra stringa che termina con \\ , quindi premete <INVIO>."
```

```
read -r var2  # L'opzione -r fa sì che "\" venga interpretata letteralmente.
              #     prima riga \

echo "var2 = $var2"
#     var2 = prima riga \

# L'introduzione dei dati termina con il primo <INVIO>.

echo

exit 0
```

Il comando **read** possiede alcune interessanti opzioni che consentono di visualizzare un prompt e persino di leggere i tasti premuti senza il bisogno di premere **INVIO**.

```
# Rilevare la pressione di un tasto senza dover premere INVIO.

read -s -n1 -p "Premi un tasto " tasto
echo; echo "Hai premuto il tasto "\"$tasto\""."

#  L'opzione -s serve a non visualizzare l'input.
#  L'opzione -n N indica che devono essere accettati solo N caratteri di input.
#  L'opzione -p permette di visualizzare il messaggio del prompt immediatamente
#+ successivo, prima di leggere l'input.

#  Usare queste opzioni è un po' complicato, perché
#+ devono essere poste nell'ordine esatto.
```

L'opzione `-n` di **read** consente anche il rilevamento dei *tasti freccia* ed alcuni altri tasti inusuali.

Example 11-5. Rilevare i tasti freccia

```
#!/bin/bash
# arrow-detect.sh: Rileva i tasti freccia, e qualcos'altro.
# Grazie a Sandro Magi per avermelo mostrato.

# --------------------------------------------------------
# Codice dei caratteri generati dalla pressione dei tasti.
frecciasu='\[A'
frecciagiù='\[B'
frecciadestra='\[C'
frecciasinistra='\[D'
inst='\[2'
canc='\[3'
# --------------------------------------------------------

SUCCESSO=0
ALTRO=65

echo -n "Premi un tasto...  "
#  Potrebbe essere necessario premere anche INVIO se viene digitato un
#+ tasto non tra quelli elencati.
```

```
read -n3 tasto                        # Legge 3 caratteri.

echo -n "$tasto" | grep "$frecciasu"  #  Verifica il codice del
                                      #+ tasto premuto.
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto Freccia-su."
  exit $SUCCESSO
fi

echo -n "$tasto" | grep "$frecciagiù"
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto Freccia-giù."
  exit $SUCCESSO
fi

echo -n "$tasto" | grep "$frecciadestra"
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto Freccia-destra."
  exit $SUCCESSO
fi

echo -n "$tasto" | grep "$frecciasinistra"
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto Freccia-sinistra."
  exit $SUCCESSO
fi

echo -n "$tasto" | grep "$ins"
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto \"Ins\"."
  exit $SUCCESSO
fi

echo -n "$tasto" | grep "$canc"
if [ "$?" -eq $SUCCESSO ]
then
  echo "È stato premuto il tasto \"Canc\"."
  exit $SUCCESSO
fi


echo " È stato premuto un altro tasto."

exit $ALTRO

#  Esercizi:
#  ---------
#  1) Semplificate lo script trasformando le verifiche multiple "if" in un
```

```
#     costrutto 'case'.
#  2) Aggiungete il rilevamento dei tasti "Home", "Fine", "PgUp" e "PgDn".

#  N.d.T. Attenzione! I codici dei tasti indicati all'inizio potrebbero non
#+ corrispondere a quelli della vostra tastiera.
#  Verificateli e quindi, modificate l'esercizio in modo che funzioni
#+ correttamente.
```

L'opzione `-t` di **read** consente un input temporizzato (vedi Example 9-4).

Il comando **read** può anche "leggere" il valore da assegnare alla variabile da un file rediretto allo `stdin`. Se il file contiene più di una riga, solo la prima viene assegnata alla variabile. Se **read** ha più di un parametro, allora ad ognuna di queste variabili vengono assegnate le stringhe successive delimitate da spazi. Attenzione!

Example 11-6. Utilizzare read con la redirezione di file

```
#!/bin/bash

read var1 <file-dati
echo "var1 = $var1"
# var1 viene impostata con l'intera prima riga del file di input "file-dati"

read var2 var3 <file-dati
echo "var2 = $var2   var3 = $var3"
# Notate qui il comportamento poco intuitivo di "read".
# 1) Ritorna all'inizio del file di input.
# 2) Ciascuna variabile viene impostata alla stringa corrispondente,
#    separata da spazi, piuttosto che all'intera riga di testo.
# 3) La variabile finale viene impostata alla parte rimanente della riga.
# 4) Se ci sono più variabili da impostare di quante siano le
#    stringhe separate da spazi nella prima riga del file, allora le
#    variabili in eccesso restano vuote.

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

# Come risolvere il problema precedente con un ciclo:
while read riga
do
  echo "$riga"
done <file-dati
# Grazie a Heiner Steven per la puntualizzazione.

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

#  Uso della variabile $IFS (Internal File Separator) per suddividere
#+ una riga di input per "read",
#+ se non si vuole che il delimitatore preimpostato sia la spaziatura.

echo "Elenco di tutti gli utenti:"
OIFS=$IFS; IFS=:       # /etc/passwd usa ":" come separatore di campo.
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
```

```
done <etc/passwd       # Redirezione I/O.
IFS=$OIFS              # Ripristina il valore originario di $IFS.
# Anche questo frammento di codice è di Heiner Steven.



#  Impostando la variabile $IFS all'interno dello stesso ciclo,
#+ viene eliminata la necessità di salvare il valore originario
#+ di $IFS in una variabile temporanea.
#  Grazie, Dim Segebart per la precisazione.
echo "-----------------------------------------------"
echo "Elenco di tutti gli utenti:"

while IFS=: read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done <etc/passwd       # Redirezione I/O.

echo
echo "\$IFS è ancora $IFS"

exit 0
```

Note: Il tentativo di impostare delle variabili collegando con una pipe l'output del comando echo a **read**, non riesce.

Tuttavia, collegare con una pipe l'output di cat *sembra* funzionare.

```
cat file1 file2 |
while read riga
do
echo $riga
done
```

Comunque, come mostra Björn Eriksson:

Example 11-7. Problemi leggendo da una pipe

```
#!/bin/sh
# readpipe.sh
# Esempio fornito da Bjon Eriksson.

ultimo="(null)"
cat $0 |
while read riga
do
    echo "{$riga}"
    ultimo=$riga
done
printf "\nFatto, ultimo:$ultimo\n"

exit 0  # Fine del codice.
        # Segue l'output (parziale) dello script.
        # 'echo' fornisce le parentesi graffe aggiuntive.
```

```
#########################################################

./readpipe.sh

{#!/bin/sh}
{ultimo="(null)"}
{cat $0 |}
{while read riga}
{do}
{echo "{$riga}"}
{ultimo=$riga}
{done}
{printf "nFatto, ultimo:$ultimon"}


Fatto, ultimo:(null)

La variabile (ultimo) è stata impostate dentro a una subshell,
quindi rimane non impostate al di fuori.
```

Filesystem

cd

Il familiare comando di cambio di directory **cd** viene usato negli script in cui, per eseguire un certo comando, è necessario trovarsi in una directory specifica.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[dal già citato esempio di Alan Cox]

L'opzione `-P` (physical) di **cd** permette di ignorare i link simbolici.

cd - cambia a $OLDPWD, la directory di lavoro precedente.

> **Caution**
>
> Il comando **cd** non funziona come ci si potrebbe aspettare quando è seguito da una doppia barra.
>
> ```
> bash$ cd //
> bash$ pwd
> //
> ```
>
> L'output, naturalmente, dovrebbe essere `/`. Questo rappresenta un problema sia da riga di comando che in uno script.

pwd

Print Working Directory. Fornisce la directory corrente dell'utente (o dello script) (vedi Example 11-8). Ha lo stesso effetto della lettura del valore della variabile builtin $PWD.

pushd
popd
dirs

Questa serie di comandi forma un sistema per tenere nota delle directory di lavoro; un mezzo per spostarsi avanti e indietro tra le directory in modo ordinato. Viene usato uno stack (del tipo LIFO) per tenere traccia dei nomi delle directory. Diverse opzioni consentono varie manipolazioni dello stack delle directory.

`pushd nome-dir` immette il percorso di *`nome-dir`* nello stack delle directory e simultaneamente passa dalla directory di lavoro corrente a *`nome-dir`*

popd preleva (pop) il nome ed il percorso della directory che si trova nella locazione più alta dello stack delle directory e contemporaneamente passa dalla directory di lavoro corrente a quella prelevata dallo stack.

dirs elenca il contenuto dello stack delle directory (lo si confronti con la variabile $DIRSTACK). Un comando **pushd** o **popd**, che ha avuto successo, invoca in modo automatico **dirs**.

Gli script che necessitano di ricorrenti cambiamenti delle directory di lavoro possono trarre giovamento dall'uso di questi comandi, evitando di dover codificare ogni modifica all'interno dello script. È da notare che nell'array implicito `$DIRSTACK`, accessibile da uno script, è memorizzato il contenuto dello stack delle directory.

Example 11-8. Cambiare la directory di lavoro corrente

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
#  Viene eseguito un 'dirs' automatico (visualizza lo stack delle
#+ directory allo stdout).
echo "Ora sei nella directory `pwd`." #  Uso degli apici singoli
                                      #+ inversi per 'pwd'.

# Ora si fa qualcosa nella directory 'dir1'.
pushd $dir2
echo "Ora sei nella directory `pwd`."

# Adesso si fa qualcos'altro nella directory 'dir2'.
echo "Nella posizione più alta dell'array DIRSTACK si trova $DIRSTACK."
popd
echo "Sei ritornato alla directory `pwd`."

# Ora si fa qualche altra cosa nella directory 'dir1'.
popd
echo "Sei tornato alla directory di lavoro originaria `pwd`."

exit 0
```

Variabili

let

Il comando **let** permette di eseguire le operazioni aritmetiche sulle variabili. In molti casi, opera come una versione meno complessa di expr.

Example 11-9. Facciamo fare a “let” qualche calcolo aritmetico.

```
#!/bin/bash

echo

let a=11            # Uguale a ’a=11’
let a=a+5           # Equivale a  let "a = a + 5"
                    # (i doppi apici e gli spazi la rendono più leggibile.)
echo "11 + 5 = $a"  # 16

let "a <<= 3"       # Equivale a  let "a = a << 3"
echo "\"\$a\" (=16) scorrimento a sinistra di 3 bit = $a"
                    # 128

let "a /= 4"        # Equivale a  let "a = a / 4"
echo "128 / 4 = $a" # 32

let "a -= 5"        # Equivale a  let "a = a - 5"
echo "32 - 5 = $a"  # 27

let "a = a * 10"    # Equivale a  let "a = a * 10"
echo "27 * 10 = $a" # 270

let "a %= 8"        # Equivale a  let "a = a % 8"
echo "270 modulo 8 = $a  (270 / 8 = 33, resto $a)"
                    # 6

echo

exit 0
```

eval

`eval arg1 [arg2] ... [argN]`

Traduce in comandi gli argomenti elencati (utile per produrre codice all’interno di uno script).

Example 11-10. Dimostrazione degli effetti di eval

```
#!/bin/bash

y=‘eval ls -l‘  #  Simile a  y=‘ls -l‘
echo $y         #  ma con i ritorni a capo tolti perché la variabile
                #+ "visualizzata" è senza "quoting".
echo
echo "$y"       #  I ritorni a capo vengono mantenuti con il
                #+ "quoting" della variabile.

echo; echo

y=‘eval df‘     # Simile a y=‘df‘
echo $y         # ma senza ritorni a capo.

#  Se non si preservano i ritorni a capo, la verifica dell’output
#+ con utility come "awk" risulta più facile.

exit 0
```

Example 11-11. Forzare un log-off

```
#!/bin/bash

y=‘eval ps ax | sed -n ’/ppp/p’ | awk ’{ print $1 }’‘
# Ricerca il numero di processo di ’ppp’.

kill -9 $y   # Lo termina

# Le righe precedenti possono essere sostituite da:
#  kill -9 ‘ps ax | awk ’/ppp/ { print $1 }’


chmod 666 /dev/ttyS3
#  Inviando un SIGKILL a ppp vengono modificati i permessi della porta seriale.
#  Vanno ripristinati allo stato precedente il SIGKILL.

rm /var/lock/LCK..ttyS3   # Cancella il lock file della porta seriale.

exit 0
```

Example 11-12. Una versione di “rot13”

```
#!/bin/bash
# Una versione di "rot13" usando ’eval’.
# Confrontatelo con l’esempio "rot13.sh".

impvar_rot_13()              # Codifica "rot13"
{
  local nomevar=$1 valoreval=$2
  eval $nomevar=’$(echo "$valoreval" | tr a-z n-za-m)’
```

```
}


impvar_rot_13 var "foobar"   # Codifica "foobar" con rot13.
echo $var                    # sbbone

echo $var | tr a-z n-za-m    # foobar
                             # Ritorno al valore originario della variabile.

# Esempio di Stephane Chazelas.

exit 0
```

Rory Winston ha fornito il seguente esempio che dimostra quanto possa essere utile **eval**.

Example 11-13. Utilizzare eval per forzare una sostituzione di variabile in uno script Perl

```
Nello script Perl  "test.pl":
        ...
        my $WEBROOT = <WEBROOT_PATH>;
        ...

Per forzare la sostituzione di variabile provate:
        $export WEBROOT_PATH=/usr/local/webroot
        $sed 's/<WEBROOT_PATH>/$WEBROOT_PATH/' < test.pl > out

Ma questo dà solamente:
        my $WEBROOT = $WEBROOT_PATH;

Tuttavia:
        $export WEBROOT_PATH=/usr/local/webroot
        $eval sed 's%\<WEBROOT_PATH\>%$WEBROOT_PATH%' < test.pl > out
#        ====

Che funziona bene, eseguendo l'attesa sostituzione:
        my $WEBROOT = /usr/local/webroot;


### Correzioni all'esempio originale eseguite da Paulo Marcel Coelho Aragao.
```

> **Caution**
>
> Il comando **eval** può essere rischioso e normalmente, quando esistono alternative ragionevoli, dovrebbe essere evitato. Un **`eval $COMANDI`** esegue tutto il contenuto di *`COMANDI`*, che potrebbe riservare spiacevoli sorprese come un **rm -rf ***. Eseguire del codice non molto familiare contenente un **eval**, e magari scritto da persone sconosciute, significa vivere pericolosamente.

set

Il comando **set** modifica il valore delle variabili interne di uno script. Un possibile uso è quello di attivare/disattivare le modalità (opzioni), legate al funzionamento della shell, che determinano il comportamento dello script. Un'altra applicazione è quella di reimpostare i parametri posizionali passati ad uno script con il risultato dell'istruzione (**`set 'comando'`**). Lo script assume i campi dell'output di comando come parametri posizionali.

Example 11-14. Utilizzare set con i parametri posizionali

```
#!/bin/bash

# script "set-test"

# Invocate lo script con tre argomenti da riga di comando,
# per esempio, "./set-test uno due tre".

echo
echo "Parametri posizionali prima di  set \‘uname -a\‘ :"
echo "Argomento nr.1 da riga di comando = $1"
echo "Argomento nr.2 da riga di comando = $2"
echo "Argomento nr.3 da riga di comando = $3"


set ‘uname -a‘  # Imposta i parametri posizionali all’output
                # del comando ‘uname -a‘

echo $_         # Sconosciuto
# Opzioni impostate nello script.

echo "Parametri posizionali dopo  set \‘uname -a\‘ :"
# $1, $2, $3, ecc. reinizializzati col risultato di ‘uname -a‘
echo "Campo nr.1 di ’uname -a’ = $1"
echo "Campo nr.2 di ’uname -a’ = $2"
echo "Campo nr.3 di ’uname -a’ = $3"
echo ---
echo $_         # ---
echo

exit 0
```

Invocando **set** senza alcuna opzione, o argomento, viene visualizzato semplicemente l'elenco di tutte le variabili d'ambiente, e non solo, che sono state inizializzate.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
BASH=/bin/bash
BASH_VERSION=$’2.05.8(1)-release’
...
XAUTHORITY=/home/bozo/.Xauthority
_=/etc/bashrc
variabile22=abc
variabile23=xzy
```

set con `--$variabile` assegna in modo esplicito il contenuto della variabile ai parametri posizionali. Se non viene specificata nessuna variabile dopo `--`, i parametri posizionali vengono *annullati*.

Example 11-15. Riassegnare i parametri posizionali

```
#!/bin/bash

variabile="uno due tre quattro cinque"

set -- $variabile
# Imposta i parametri posizionali al contenuto di "$variabile".

primo_param=$1
secondo_param=$2
shift; shift        # Salta i primi due parametri posizionali.
restanti_param="$*"

echo
echo "primo parametro = $primo_param"             # uno
echo "secondo parametro = $secondo_param"         # due
echo "rimanenti parametri = $restanti_param"      # tre quattro cinque

echo; echo

# Ancora.
set -- $variabile
primo_param=$1
secondo_param=$2
echo "primo parametro = $primo_param"             # uno
echo "secondo parametro = $secondo_param"         # due

# ======================================================

set --
#  Annulla i parametri posizionali quando non viene specificata
#+ nessuna variabile.

primo_param=$1
secondo_param=$2
echo "primo parametro = $primo_param"             # (valore nullo)
echo "secondo parametro = $secondo_param"         # (valore nullo)

exit 0
```

Vedi anche Example 10-2 e Example 12-41.

unset

il comando **unset** annulla una variabile di shell, vale a dire, la imposta al valore *nullo*. Fate attenzione che questo comando non è applicabile ai parametri posizionali.

```
bash$ unset PATH

bash$ echo $PATH


bash$
```

Example 11-16. “Annullare” una variabile

```
#!/bin/bash
# unset.sh: Annullare una variabile.

variabile=ciao                             # Inizializzata.
echo "variabile = $variabile"

unset variabile                            # Annullata.
                                           # Stesso effetto di variabile=
echo "variabile (annullata) = $variabile"  # $variabile è nulla.

exit 0
```

export

Il comando **export** rende disponibili le variabili a tutti i processi figli generati dallo script in esecuzione o dalla shell. Purtroppo, non vi è alcun modo per **esportare** le variabili in senso contrario verso il processo genitore, ovvero nei confronti del processo che ha chiamato o invocato lo script o la shell. Un uso importante del comando **export** si trova nei file di avvio (startup) per inizializzare e rendere accessibili le variabili d’ambiente ai susseguenti processi utente.

Example 11-17. Utilizzare export per passare una variabile ad uno script awk incorporato

```
#!/bin/bash

#  Ancora un’altra versione dello script "totalizzatore di colonna"
#+ (col-totaler.sh) che aggiunge una specifica colonna (di numeri)
#+ nel file di destinazione. Viene sfruttato l’ambiente per passare
#+ una variabile dello script ’awk’.

ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ] #  Verifica il corretto numero di argomenti da
                     #+ riga di comando.
then
   echo "Utilizzo: ‘basename $0‘ nomefile colonna-numero"
```

```
   exit $E_ERR_ARG
fi

nomefile=$1
colonna_numero=$2

#===== Fino a questo punto è uguale allo script originale =====#

export colonna_numero
# Esporta il numero di colonna all'ambiente, in modo che sia disponibile
#+ all'utilizzo.


# Inizio dello script awk.
# ------------------------------------------------
awk '{ totale += $ENVIRON["colonna_numero"]
}
END { print totale }' $nomefile
# ------------------------------------------------
# Fine dello script awk.


# Grazie, Stephane Chazelas.

exit 0
```

> **Tip:** È possibile inizializzare ed esportare variabili con un'unica operazione, come **export var1=xxx**.
>
> Tuttavia, come ha sottolineato Greg Keraunen, in certe situazioni questo può avere un effetto diverso da quello che si avrebbe impostando prima la variabile ed esportandola successivamente.
>
> ```
> bash$ export var=(a b); echo ${var[0]}
> (a b)
>
>
>
> bash$ var=(a b); export var; echo ${var[0]}
> a
> ```

declare
typeset

I comandi declare e typeset specificano e/o limitano le proprietà delle variabili.

readonly

Come declare -r, imposta una variabile in sola lettura ovvero, in realtà, come una costante. I tentativi per modificare la variabile falliscono con un messaggio d'errore. È l'analogo shell del qualificatore di tipo **const** del linguaggio C.

getopts

Questo potente strumento verifica gli argomenti passati da riga di comando allo script. È l'analogo Bash del comando esterno getopt e della funzione di libreria **getopt** familiare ai programmatori in C. Permette di passare e concatenare più opzioni [2] e argomenti associati allo script (per esempio **`nomescript -abc -e /usr/local`**).

Il costrutto **getopts** utilizza due variabili implicite. `$OPTIND`, che è il puntatore all'argomento, (*OPTion INDex*) e `$OPTARG` (*OPTion ARGument*) l'argomento (eventuale) associato ad un'opzione. Nella dichiarazione, i due punti che seguono il nome dell'opzione servono ad identificare quell'opzione come avente un argomento associato.

Il costrutto **getopts** di solito si trova all'interno di un ciclo while, che elabora le opzioni e gli argomenti uno alla volta e quindi decrementa la variabile implicita `$OPTIND` per il passo successivo.

Note:

1. Gli argomenti passati allo script da riga di comando devono essere preceduti da un meno (-) o un più (+). È il prefisso - o + che consente a **getopts** di riconoscere gli argomenti da riga di comando come *opzioni*. Infatti **getopts** non elabora argomenti che non siano preceduti da - o + e termina la sua azione appena incontra un'opzione che ne è priva.
2. La struttura di **getopts** differisce leggermente da un normale ciclo **while** perché non è presente la condizione di verifica.
3. Il costrutto **getopts** sostituisce il meno potente ed obsoleto comando esterno getopt.

```
while getopts ":abcde:fg" Opzione
# Dichiarazione iniziale.
# a, b, c, d, e, f, g sono le opzioni attese.
# I : dopo l'opzione 'e' indicano che c'è un argomento associato.
do
  case $Opzione in
  a ) # Fa qualcosa con la variabile 'a'.
  b ) # Fa qualcosa con la variabile 'b'.
  ...
  e)  # Fa qualcosa con 'e', e anche con $OPTARG,
      # che è l'argomento associato all'opzione 'e'.
  ...
  g ) # Fa qualcosa con la variabile 'g'.
  esac
done
shift $(($OPTIND - 1))
# Sposta il puntatore dell'argomento al successivo.

# Tutto questo non è affatto complicato come sembra <sorriso>.
```

Example 11-18. Utilizzare getopts per leggere le opzioni o gli argomenti passati ad uno script

```
#!/bin/bash
# Prove con getopts e OPTIND
# Script modificato il 9/10/03 su suggerimento di Bill Gradwohl.

#  Osserviamo come 'getopts' elabora gli argomenti passati allo script da
#+ riga di comando.
#  Gli argomenti vengono verificati come "opzioni" (flag)
#+ ed argomenti associati.

#  Provate ad invocare lo script con
#  'nomescript -mn'
#  'nomescript -oq qOpzione' (qOpzione può essere una stringa qualsiasi.)
#  'nomescript -qXXX -r'
#
#  'nomescript -qr'    - Risultato inaspettato, considera "r"
#+ come l'argomento dell'opzione "q"
#  'nomescript -q -r'  - Risultato inaspettato, come prima.
#  'nomescript -mnop -mnop'  - Risultato inaspettato
#  (OPTIND non è attendibile nello stabilire da dove proviene un'opzione).
#
#  Se un'opzione si aspetta un argomento ("flag:"), viene presa
#  qualunque cosa si trovi vicino.

NO_ARG=0
E_ERR_OPZ=65

if [ $# -eq "$NO_ARG" ]   #  Lo script è stato invocato senza
                          #+ alcun argomento?
then
  echo "Utilizzo: 'basename $0' opzioni (-mnopqrs)"
  exit $E_ERR_OPZ         #  Se non ci sono argomenti, esce e
                          #+ spiega come usare lo script.
fi
# Utilizzo: nomescript -opzioni
# Nota: è necessario il trattino (-)


while getopts ":mnopq:rs" Opzione
do
  case $Opzione in
    m     ) echo "Scenario nr.1: opzione -m-  [OPTIND=${OPTIND}]";;
    n | o ) echo "Scenario nr.2: opzione -$Opzione-  [OPTIND=${OPTIND}]";;
    p     ) echo "Scenario nr.3: opzione -p-  [OPTIND=${OPTIND}]";;
    q     ) echo "Scenario nr.4: opzione -q-\
con argomento \"$OPTARG\"  [OPTIND=${OPTIND}]";;
    # Notate che l'opzione 'q' deve avere un argomento associato,
    # altrimenti salta alla voce predefinita del costrutto case.
    r | s ) echo "Scenario nr.5: opzione -$Opzione-"";;
    *     ) echo "È stata scelta un'opzione non implementata.";; # DEFAULT
  esac
done
```

```
shift $(($OPTIND - 1))
#  Decrementa il puntatore agli argomenti in modo che punti al successivo.
#  $1 fa ora riferimento al primo elemento non opzione fornito da riga di
#+ comando, ammesso che ci sia.

exit 0

#   Come asserisce Bill Gradwohl,
#  "Il funzionamento di getopts permette di specificare: nomescript -mnop -mnop,
#+  ma non esiste, utilizzando OPTIND, nessun modo affidabile per differenziare
#+  da dove proviene che cosa."
```

Comportamento dello Script

source
. (punto comando)

Questa istruzione, se invocata da riga di comando, esegue uno script. All'interno di uno script, **`source nome-file`** carica il file `nome-file`. Nello scripting di shell, questo è l'equivalente della direttiva **`#include`** del C/C++. È utile in situazioni in cui diversi script usano un file dati comune o una stessa libreria di funzioni.

Example 11-19. "Includere" un file dati

```
#!/bin/bash

. file-dati    # Carica un file dati.
# Stesso effetto di "source file-dati", ma più portabile.
#  Il file "file-dati" deve essere presente nella directory di lavoro
#+ corrente, poiché vi si fa riferimento per mezzo del suo 'basename'.

# Ora utilizziamo alcuni dati del file.

echo "variabile1 (dal file-dati) = $variabile1"
echo "variabile3 (dal file-dati) = $variabile3"

let "sommma = $variabile2 + $variabile4"
echo "Somma della variabile2 + variabile4 (dal file-dati) = $somma"
echo "messaggio1 (dal file-dati)  \"$messaggio1\""
# Nota:                            apici doppi con escape.

visualizza_messaggio Questa è la funzione di visualizzazione messaggio \
presente in file-dati.


exit 0
```

Il file `file-dati` per l'Example 11-19 precedente. Dev'essere presente nella stessa directory.

```
#  Questo è il file dati caricato dallo script.
#  File di questo tipo possono contenere variabili, funzioni, ecc.
```

```
#  Può essere caricato con il comando ’source’ o ’.’ da uno script di shell.

# Inizializziamo alcune variabili.

variabile1=22
variabile2=474
variabile3=5
variabile4=97

messaggio1="Ciao, come stai?"
messaggio2="Per ora piuttosto bene. Arrivederci."

visualizza_messaggio ()
{
# Visualizza qualsiasi messaggio passato come argomento.

  if [ -z "$1" ]
  then
  return 1
  # Errore, se l’argomento è assente.
  fi

  echo

  until [ -z "$1" ]
  do
  # Scorre gli argomenti passati alla funzione.
    echo -n "$1"
    #  Visualizza gli argomenti uno alla volta, eliminando i ritorni a capo.
    echo -n " "
    # Inserisce degli spazi tra le parole.
    shift
    # Successivo.
  done

  echo

  return 0
}
```

È anche possibile per uno script usare *source* in riferimento a se stesso, sebbene questo non sembri avere reali applicazioni pratiche.

Example 11-20. Un (inutile) script che esegue se stesso

```
#!/bin/bash
# self-source.sh: uno script che segue se stesso "ricorsivamente."
# Da "Stupid Script Tricks," Volume II.

MAXPASSCNT=100    # Numero massimo di esecuzioni.

echo -n  "$conta_passi  "
#  Al primo passaggio, vengono visualizzati solo due spazi,
```

```
#+ perché $conta_passi non è stata inizializzata.

let "conta_passi += 1"
#  Si assume che la variabile $conta_passi non inizializzata possa essere
#+ incrementata subito.
#  Questo funziona con Bash e pdksh, ma si basa su un'azione non portabile
#+ (e perfino pericolosa).
#  Sarebbe meglio impostare $conta_passi a 0 nel caso non fosse stata
#+ inizializzata.

while [ "$conta_passi" -le $MAXPASSCNT ]
do
  . $0   #  Lo script "esegue" se stesso, non chiama se stesso.
         #  ./$0 (che sarebbe la vera ricorsività) in questo caso non funziona.
done

#  Quello che avviene in questo script non è una vera ricorsività, perché lo
#+ script in realtà "espande" se stesso (genera una nuova sezione di codice)
#+ ad ogni passaggio attraverso il ciclo 'while', con ogni 'source' che si
#+ trova alla riga 20.
#
#  Naturalmente, lo script interpreta ogni succesiva 'esecuzione' della riga
#+ con "#!" come un commento e non come l'inizio di un nuovo script.

echo

exit 0   # Il risultato finale è un conteggio da 1 a 100.
         # Molto impressionante.

# Esercizio:
# --------
#  Scrivete uno script che usi questo espediente per fare qualcosa di utile.
```

exit

Termina in maniera incondizionata uno script. Il comando **exit** opzionalmente può avere come argomento un intero che viene restituito alla shell come exit status dello script. È buona pratica terminare tutti gli script, tranne quelli più semplici, con **`exit 0`**, indicandone con ciò la corretta esecuzione.

> **Note:** Se uno script termina con un **exit** senza argomento, l'exit status dello script corrisponde a quello dell'ultimo comando eseguito nello script, escludendo **exit**.

exec

Questo builtin di shell sostituisce il processo corrente con un comando specificato. Normalmente, quando la shell incontra un comando, genera (forking) un processo figlio che è quello che esegue effettivamente il comando. Utilizzando il builtin **exec**, la shell non esegue il forking ed il comando lanciato con exec sostituisce

la shell. Se viene usato in uno script ne forza l'uscita quando il comando eseguito con **exec** termina. Per questa ragione, se in uno script è presente **exec**, probabilmente questo sarà il comando finale.

Example 11-21. Effetti di exec

```
#!/bin/bash

exec echo "Uscita da \"$0\"." # Esce dallo script in questo punto.

# -------------------------------------------
# Le righe seguenti non verranno mai eseguite.

echo "Questo messaggio non verrà mai visualizzato."

exit 99          #  Questo script non termina qui.
                 #  Verificate l'exit status, dopo che lo script è
                 #+ terminato, con 'echo $?'.
                 #  *Non* sarà 99.
```

Example 11-22. Uno script che esegue se stesso con exec

```
#!/bin/bash
# self-exec.sh

echo

echo "Sebbene questa riga compaia UNA SOLA VOLTA nello script, continuerà"
echo "ad essere visualizzata."
echo "Il PID di questo script d'esempio è ancora $$."
#     Dimostra che non viene generata una subshell.

echo "=================== Premi Ctl-C per uscire ==================="

sleep 1

exec $0   #  Inizia un'altra istanza di questo stesso script
          #+ che sostituisce quella precedente.

echo "Questa riga non verrà mai visualizzata"  # Perché?

exit 0
```

exec serve anche per riassegnare i descrittori dei file. **`exec <zzz-file`** sostituisce lo `stdin` con il file `zzz-file` (vedi Example 16-1).

> **Note:** L'opzione `-exec` di find *non* è la stessa cosa del builtin di shell **exec**.

shopt

Questo comando permette di cambiare le opzioni di shell al volo (vedi Example 24-1 e Example 24-2). Appare spesso nei file di avvio (startup) Bash, ma può essere usato anche in altri script. È necessaria la versione 2 o seguenti di Bash.

```
shopt -s cdspell
#  Consente le errate digitazioni, non gravi, dei nomi delle directory quando
#+ si usa 'cd'

cd /hpme  # Oops! Errore '/home'.
pwd       # /home
          # La shell ha corretto l'errore di digitazione.
```

Comandi

true

Comando che restituisce zero come exit status di una corretta esecuzione, ma nient'altro.

```
# Ciclo infinito
while true   # alternativa a ":"
do
   operazione-1
   operazione-2
   ...
   operazione-n
   # Occorre un sistema per uscire dal ciclo, altrimenti lo script si blocca.
done
```

false

Comando che restituisce l'exit status di una esecuzione non andata a buon fine, ma nient'altro.

```
# Ciclo che non viene eseguito
while false
do
   # Il codice seguente non verrà eseguito.
   operazione-1
   operazione-2
   ...
   operazione-n
   # Non succede niente!
done
```

type [comando]

Simile al comando esterno which, **type comando** fornisce il percorso completo di "comando". A differenza di **which**, **type** è un builtin di Bash. L'utile opzione `-a` di **type** identifica le *`parole chiave`* ed i *`builtin`*, individuando anche i comandi di sistema che hanno gli stessi nomi.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[
```

hash [comandi]

Registra i percorsi assoluti dei comandi specificati (nella tabella degli hash della shell), in modo che la shell o lo script non avranno bisogno di cercare `$PATH` nelle successive chiamate di quei comandi. Se **hash** viene eseguito senza argomenti, elenca semplicemente i comandi presenti nella tabella. L'opzione `-r` cancella la tabella degli hash.

help

help COMANDO mostra un breve riepilogo dell'utilizzo del builtin di shell COMANDO. È il corrispettivo di whatis, ma solo per i builtin.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

11.1. Comandi di controllo dei job

Alcuni dei seguenti comandi di controllo di job possono avere come argomento un "identificatore di job". Vedi la tabella alla fine del capitolo.

jobs

Elenca i job in esecuzione in background, fornendo il rispettivo numero. Non è così utile come **ps**.

> **Note:** È facilissimo confondere *job* e *processi*. Alcuni builtin, quali **kill**, **disown** e **wait**, accettano come argomento sia il numero di job che quello di processo. I comandi **fg**, **bg** e **jobs** accettano solo il numero di job.

```
bash$ sleep 100 &
[1] 1384
```

```
bash $ jobs
[1]+  Running                 sleep 100 &
```

"1" è il numero di job (i job sono gestiti dalla shell corrente), mentre "1384" è il numero di processo (i processi sono gestiti dal sistema operativo). Per terminare questo job/processo si può utilizzare sia **kill %1** che **kill 1384**.

Grazie, S.C.

disown

Cancella il/i job dalla tabella dei job attivi della shell.

fg
bg

Il comando **fg** modifica l'esecuzione di un job da background (sfondo) in foreground (primo piano). Il comando **bg** fa ripartire un job che era stato sospeso, mettendolo in esecuzione in background. Se non viene specificato nessun numero di job, allora il comando **fg** o **bg** agisce sul job attualmente in esecuzione.

wait

Arresta l'esecuzione dello script finché tutti i job in esecuzione in background non sono terminati, o finché non è terminato il job o il processo il cui numero, o id, è stato passato come opzione. Restituisce l'exit status di attesa-comando.

Il comando **wait** può essere usato per evitare che uno script termini prima che un job in esecuzione in background abbia ultimato il suo compito (ciò creerebbe un temibile processo orfano).

Example 11-23. Attendere la fine di un processo prima di continuare

```
#!/bin/bash

ROOT_UID=0   # Solo gli utenti con $UID 0 posseggono i privilegi di root.
E_NONROOT=65
E_NOPARAM=66

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Bisogna essere root per eseguire questo script."
  # "Cammina ragazzo, hai finito di poltrire."
  exit $E_NONROOT
fi

if [ -z "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ nome-cercato"
  exit $E_NOPARAM
fi
```

```
echo "Aggiornamento del database 'locate' ..."
echo "Questo richiede un po' di tempo."
updatedb /usr &     # Deve essere eseguito come root.

wait
#  Non viene eseguita la parte restante dello script finché 'updatedb' non
#+ ha terminato il proprio compito.
#  Si vuole che il database sia aggiornato prima di cercare un nome di file.

locate $1

#  Senza il comando wait, nell'ipotesi peggiore, lo script sarebbe uscito
#+ mentre 'updatedb' era ancora in esecuzione, trasformandolo in un processo
#+ orfano.

exit 0
```

Opzionalmente, **wait** può avere come argomento un identificatore di job, per esempio, **wait%1** o **wait $PPID**. Vedi la tabella degli identificatori di job.

> **Tip:** In uno script, far eseguire un comando in background, per mezzo della E commerciale (&), può causare la sospensione dello script finché non viene premuto il tasto **INVIO**. Questo sembra capitare con i comandi che scrivono allo `stdout`. Può rappresentare un grande fastidio.
>
> ```
> #!/bin/bash
> # test.sh
>
> ls -l &
> echo "Fatto."
> ```
>
> ```
> bash$ ./test.sh
> Fatto.
>  [bozo@localhost test-scripts]$ total 1
>  -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
>  _
> ```
>
> Mettendo **wait** dopo il comando che deve essere eseguito in background si rimedia a questo comportamento.
>
> ```
> #!/bin/bash
> # test.sh
>
> ls -l &
> echo "Fatto."
> wait
> ```
>
> ```
> bash$ ./test.sh
> Fatto.
>  [bozo@localhost test-scripts]$ total 1
>  -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
> ```
>
> Un altro modo per far fronte a questo problema è quello di redirigere l'output del comando in un file o anche in `/dev/null`.

suspend

Ha un effetto simile a **Control**-**Z**, ma sospende la shell (il processo genitore della shell può, ad un certo momento stabilito, farle riprendere l'esecuzione).

logout

È il comando di uscita da una shell di login. Facoltativamente può essere specificato un exit status.

times

Fornisce statistiche sul tempo di sistema impiegato per l'esecuzione dei comandi, nella forma seguente:

```
0m0.020s 0m0.020s
```

Questo comando ha un valore molto limitato perché non è di uso comune tracciare profili o benchmark degli script di shell.

kill

Termina immediatamente un processo inviandogli un appropriato segnale di *terminazione* (vedi Example 13-4).

Example 11-24. Uno script che uccide se stesso

```
#!/bin/bash
# self-destruct.sh

kill $$  # Lo script in questo punto "uccide" il suo stesso processo.
         # Ricordo che "$$" è il PID dello script.

echo "Questa riga non viene visualizzata."
# Invece, la shell invia il messaggio "Terminated" allo stdout.

exit 0

#  Dopo che lo script è terminato prematuramente, qual'è l'exit
#+ status restituito?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#             segnale SIGTERM
```

> **Note:** **`kill -l`** elenca tutti i segnali. **`kill -9`** è il "killer infallibile", che solitamente interrompe un processo che si rifiuta ostinatamente di terminare con un semplice **kill**. Talvolta funziona anche **`kill -15`**. Un "processo zombie", vale a dire un processo il cui genitore è stato terminato, non può essere ucciso (non si può uccidere qualcosa che è già morto). Comunque **init** presto o tardi, solitamente lo cancellerà.

command

La direttiva **command COMANDO** disabilita gli alias e le funzioni del comando "COMANDO".

> **Note:** È una delle tre direttive di shell attinenti all'elaborazione dei comandi di uno script. Le altre sono builtin ed enable.

builtin

Invocando **builtin COMANDO_BUILTIN** viene eseguito "COMANDO_BUILTIN" come se fosse un builtin di shell, disabilitando temporaneamente sia le funzioni che i comandi di sistema esterni aventi lo stesso nome.

enable

Abilita o disabilita un builtin di shell. Ad esempio, **enable -n kill** disabilita il builtin di shell kill, così quando Bash successivamente incontra un **kill**, invocherà `/bin/kill`.

L'opzione `-a` di **enable** elenca tutti i builtin di shell, indicando se sono abilitati o meno. L'opzione `-f nomefile` permette ad **enable** di caricare un builtin come un modulo di una libreria condivisa (DLL) da un file oggetto correttamente compilato. [3].

autoload

È un adattamento per Bash dell'autoloader *ksh*. In presenza di un **autoload** , viene caricata una funzione contenente una dichiarazione "autoload" da un file esterno, alla sua prima invocazione. [4] Questo risparmia risorse di sistema.

È da notare che **autoload** non fa parte dell'installazione normale di Bash. Bisogna caricarlo con **enable -f** (vedi sopra).

Table 11-1. Identificatori di job

Notazione	Significato
`%N`	numero associato al job [N]
`%S`	Chiamata (da riga di comando) del job che inizia con la stringa *S*
`%?S`	Chiamata (da riga di comando) del job con al suo interno la stringa *S*
`%%`	job "corrente" (ultimo job arrestato in foreground o iniziato in background)
`%+`	job "corrente" (ultimo job arrestato in foreground o iniziato in background)
`%-`	Ultimo job

Notazione	Significato
`$!`	Ultimo processo in background

Notes

1. Un’eccezione è rappresentata dal comando time, citato nella documentazione ufficiale Bash come parola chiave.
2. Un’opzione è un argomento che funziona come un interruttore, attivando/disattivando le modalità di azione di uno script. L’argomento associato ad una particolare opzione ne indica o meno l’abilitazione.
3. I sorgenti C per un certo numero di builtin caricabili, solitamente, si trovano nella directory `/usr/share/doc/bash-?.??/functions`.

   E’ da notare che l’opzione `-f` di **enable** non è portabile su tutti i sistemi.
4. Lo stesso risultato di **autoload** può essere ottenuto con typeset -fu.

Chapter 12. Filtri, programmi e comandi esterni

I comandi standard UNIX rendono gli script di shell più versatili. La potenza degli script deriva dall'abbinare, in semplici costrutti di programmazione, comandi di sistema e direttive di shell.

12.1. Comandi fondamentali

I primi comandi che il neofita deve conoscere

ls

Il comando fondamentale per "elencare" i file. È molto facile sottostimare la potenza di questo umile comando. Per esempio, l'uso dell'opzione `-R`, ricorsivo, con **ls** provvede ad elencare la directory in forma di struttura ad albero. Altre interessanti opzioni sono: `-S`, ordina l'elenco in base alla dimensione, `-t`, lo ordina in base alla data di modifica e `-i` che mostra gli inode dei file (vedi Example 12-3).

Example 12-1. Utilizzare ls per creare un sommario da salvare in un CDR

```
#!/bin/bash
# burn-cd.sh
# Script per rendere automatica la registrazione di un CDR.

VELOC=2          #  Potete utilizzare una velocità più elevata
                 #+ se l'hardware la supporta.
FILEIMMAGINE=cdimage.iso
CONTENUTIFILE=contenuti
DEFAULTDIR=/opt  # Questa è la directory contenente i dati da registrare.
                 # Accertatevi che esista.

# Viene usato il pacchetto  "cdrecord" di Joerg Schilling.
# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

#  Se questo script viene eseguito da un utente ordinario va impostato
#+ il bit suid di cdrecord (chmod u+s /usr/bin/cdrecord, da root).

if [ -z "$1" ]
then
  DIRECTORY_IMMAGINE=$DEFAULTDIR
  #  Viene usata la directory predefinita se non ne viene specificata
  #+ alcuna da riga di comando.
else
  DIRECTORY_IMMAGINE=$1
fi

#  Crea il "sommario" dei file.
ls -lRF $DIRECTORY_IMMAGINE > $DIRECTORY_IMMAGINE/$CONTENUTIFILE
#  L'opzione "l" fornisce un elenco "dettagliato".
#  L'opzione "R" rende l'elencazione ricorsiva.
#  L'opzione "F" evidenzia i tipi di file (le directory hanno una
#+ "/" dopo il nome).
```

```
echo "Il sommario è stato creato."

# Crea l'immagine del file che verrà registrato sul CDR.
mkisofs -r -o $FILEIMMAGINE $DIRECTORY_IMMAGINE
echo "È stata creata l'immagine ($FILEIMMAGINE) su file system ISO9660."

# Registra il CDR.
cdrecord -v -isosize speed=$VELOC dev=0,0 $FILEIMMAGINE
echo "Sto \"bruciando\" il CD."
echo "Siate pazienti, occorre un po' di tempo."

exit 0
```

cat
tac

cat è l'acronimo di *concatenato*, visualizza un file allo `stdout`. In combinazione con gli operatori di redirezione (> o >>) è comunemente usato per concatenare file.

```
# Usi di 'cat'
cat nomefile                          # Visualizza il contenudo del file.

cat file.1 file.2 file.3 > file.123   # Concatena tre file in uno.
```

L'opzione `-n` di **cat** numera consecutivamente le righe del/dei file di riferimento. L'opzione `-b` numera solo le righe non vuote. L'opzione `-v` visualizza i caratteri non stampabili, usando la notazione ^ . L'opzione `-s` comprime tutte le righe vuote consecutive in un'unica riga vuota.

Vedi anche Example 12-22 e Example 12-18.

tac è l'inverso di *cat* e visualizza un file in senso contrario, vale a dire, partendo dalla fine.

rev

inverte ogni riga di un file e la visualizza allo `stdout`. Non ha lo stesso effetto di **tac** poiché viene preservato l'ordine delle righe, rovescia semplicemente ciascuna riga.

```
bash$ cat file1.txt
Questa è la riga 1.
Questa è la riga 2.


bash$ tac file1.txt
Questa è la riga 2.
Questa è la riga 1.


bash$ rev file1.txt
.1 agir al è atseuQ
.2 agir al è atseuQ
```

cp

È il comando per la copia dei file. **`cp file1 file2`** copia `file1` in `file2`, sovrascrivendo `file2` nel caso esistesse già (vedi Example 12-5).

> **Tip:** Sono particolarmente utili le opzioni `-a` di archiviazione (per copiare un intero albero di directory), `-r` e `-R` di ricorsività.

mv

È il comando per lo spostamento di file. È equivalente alla combinazione di **cp** e **rm**. Può essere usato per spostare più file in una directory o anche per rinominare una directory. Per alcune dimostrazioni sull'uso di **mv** in uno script, vedi Example 9-17 e Example A-3.

> **Note:** Se usato in uno script non interattivo, **mv** vuole l'opzione `-f` (*forza*) per evitare l'input dell'utente.
>
> Quando una directory viene spostata in un'altra preesistente, diventa la sottodirectory di quest'ultima.

```
bash$ mv directory_iniziale directory_destinazione

bash$ ls -lF directory_destinazione
total 1
drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 directory_iniziale/
```

rm

Cancella (rimuove) uno o più file. L'opzione `-f` forza la cancellazione anche dei file in sola lettura. È utile per evitare l'input dell'utente in uno script.

> **Warning**
>
> Se usato con l'opzione di ricorsività `-r`, il comando cancella tutti i file della directory.

rmdir

Cancella una directory. Affinché questo comando funzioni è necessario che la directory non contenga alcun file, neanche i cosiddetti "dotfile" [1].

mkdir

Crea una nuova directory. Per esempio, **`mkdir -p progetto/programmi/Dicembre`** crea la directory indicata. L'opzione `-p` crea automaticamente tutte le necessarie directory indicate nel percorso.

chmod

Modifica gli attributi di un file esistente (vedi Example 11-11).

```
chmod +x nomefile
# Rende eseguibile "nomefile" per tutti gli utenti.

chmod u+s nomefile
#  Imposta il bit "suid" di "nomefile".
#  Un utente comune può eseguire "nomefile" con gli stessi privilegi del
#+ proprietario del file (Non è applicabile agli script di shell).


chmod 644 nomefile
#  Dà al proprietario i permessi di lettura/scrittura su "nomefile", il
#+ permesso di sola lettura a tutti gli altri utenti
#  (modalità ottale).


chmod 1777 nome-directory
#  Dà a tutti i permessi di lettura, scrittura ed esecuzione nella
#+ directory, inoltre imposta lo "sticky bit". Questo significa che solo il
#+ proprietario della directory, il proprietario del file e, naturalmente, root
#+ possono cancellare dei file particolari presenti in quella directory.
```

chattr

Modifica gli attributi del file. Ha lo stesso effetto di **chmod**, visto sopra, ma con una sintassi diversa, e funziona solo su un filesystem di tipo *ext2*.

ln

Crea dei link a file esistenti. Un "link" è un riferimento a un file, un nome alternativo. Il comando **ln** permette di fare riferimento al file collegato (linkato) con più di un nome e rappresenta un'alternativa di livello superiore all'uso degli alias (vedi Example 4-6).

ln crea semplicemente un riferimento, un puntatore al file, che occupa solo pochi byte.

Il comando **ln** è usato molto spesso con l'opzione `-s`, simbolico o "soft". Uno dei vantaggi dell'uso dell'opzione `-s` è che consente riferimenti attraverso tutto il filesystem.

La sintassi del comando è un po' ingannevole. Per esempio: **`ln -s vecchiofile nuovofile`** collega `nuovofile`, creato con l'istruzione, all'esistente `vecchiofile`.

> **Caution**
>
> Nel caso sia già presente un file di nome `nuovofile`, questo verrà cancellato quando `nuovofile` diventa il nome del collegamento (link).

Quale tipo di link usare?

Ecco come lo spiega John Macdonald:

Entrambi i tipi forniscono uno strumento sicuro di referenziazione doppia -- se si edita il contenuto del file usando uno dei due nomi, le modifiche riguarderanno sia il file con il nome originario che quello con il nome nuovo, sia esso un link simbolico o hard. Le loro differenze si evidenziano quando si opera ad un livello superiore. Il vamtaggio di un hard link è che il nuovo nome è completamente indipendente da quello vecchio -- se si cancella o rinomina il vecchio file, questo non avrà alcun effetto su un hard link, che continua a puntare ai dati, mentre lascerebbe in sospeso un link simbolico che punta al vecchio nome che non esiste più. Il vantaggio di un link simbolico è che può far riferimento ad un diverso filesystem (dal momento che si tratta di un semplice collegamento al nome di un file, non ai dati reali).

man
info

Questi comandi danno accesso alle informazioni e alle pagine di manuale dei comandi di sistema e delle utility installate. Quando sono disponibili, le pagine *info*, di solito, contengono una descrizione più dettagliata che non le pagine di *manuale*.

12.2. Comandi complessi

Comandi per utenti avanzati

find

-exec *COMANDO* \;

Esegue *COMANDO* su ogni file verificato da **find**. La sequenza dei comandi termina con \; (il ";" deve essere preceduto dal carattere di escape per essere certi che la shell lo passi a **find** col suo significato letterale). Se *COMANDO* contiene {}, allora **find** sostituisce "{}" con il percorso completo del file selezionato.

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

```
find /home/bozo/projects -mtime 1
#  Elenca tutti i file della directory /home/bozo/projects
#+ che sono stati modificati il giorno precedente.
```

```
#
#  mtime = ora dell'ultima modifica del file in questione
#  ctime = ora dell'ultima modifica di stato (tramite 'chmod' o altro)
#  atime = ora dell'ultimo accesso

DIR=/home/bozo/junk_files
find "$DIR" -type f -atime +5 -exec rm {} \;
#  Cancella tutti il file in "/home/bozo/junk_files"
#+ a cui non si è acceduto da almeno 5 giorni.
#
#  "-type tipofile", dove
#  f = file regolare
#  d = directory, ecc.
#  (La pagina di manuale di 'find' contiene l'elenco completo.)


find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;

#  Trova tutti gli indirizzi IP (xxx.xxx.xxx.xxx) nei file della directory /etc.
#  Ci sono alcuni caratteri non essenziali - come possono essere tolti?

# Ecco una possibilità:

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'

# [:digit:] è una delle classi di caratteri
# introdotta con lo standard POSIX 1003.2.

# Grazie, S.C.
```

Note: L'opzione `-exec` di **find** non deve essere confusa con il builtin di shell exec.

Example 12-2. Badname, elimina, nella directory corrente, i file i cui nomi contengono caratteri inappropriati e spazi

```
#!/bin/bash

# Cancella i file nella directory corrente contenenti caratteri inadatti.

for nomefile in *
do
nomestrano=‘echo "$nomefile" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p‘
# Nomi di file contenenti questi caratteri:   + { ; " \ = ? ~ ( ) < > & * | $
rm $nomestrano 2>/dev/null    # Vengono eliminati i messaggi d'errore.
done

# Ora ci occupiamo dei file contenenti ogni tipo di spaziatura.
find . -name "* *" -exec rm -f {} \;
```

```
#  Il percorso del file che "find" cerca prende il posto di "{}".
#  La ’\’ assicura che il ’;’ sia interpretato correttamente come fine del
#+ comando.

exit 0

#-----------------------------------------------------------------------
# I seguenti comandi non vengono eseguiti a causa dell’"exit" precedente.

# Un’alternativa allo script visto prima:
find . -name ’*[+{;"\\=?~()<>&*|$ ]*’ -exec rm -f ’{}’ \;
exit 0
# (Grazie, S.C.)
```

Example 12-3. Cancellare un file tramite il suo numero di *inode*

```
#!/bin/bash
# idelete.sh: Cancellare un file per mezzo del suo numero di inode.

#  Questo si rivela utile quando il nome del file inizia con un
#+ carattere non permesso, come ? o -.

ARGCONTO=1          #  Allo script deve essere passato come argomento
                    #+ il nome del file.
E_ERR_ARG=70
E_FILE_NON_ESISTE=71
E_CAMBIO_IDEA=72

if [ $# -ne "$ARGCONTO" ]
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG
fi

if [ ! -e "$1" ]
then
  echo "Il file \""$1"\" non esiste."
  exit $E_FILE_NON_ESISTE
fi

inum=‘ls -i | grep "$1" | awk ’{print $1}’‘
#  inum = numero di inode (index node) del file
#  Tutti i file posseggono un inode, la registrazione che contiene
#+ informazioni sul suo indirizzo fisico.

echo; echo -n "Sei assolutamente sicuro di voler cancellare \"$1\"(s/n)?"
# Anche ’rm’ con l’opzione ’-v’ visualizza la stessa domanda.
read risposta
case "$risposta" in
[nN]) echo "Hai cambiato idea, vero?"
      exit $E_CAMBIO_IDEA
      ;;
```

```
*)    echo "Cancello il file \"$1\".";;
esac

find . -inum $inum -exec rm {} \;
echo "Il file "\"$1"\" è stato cancellato!"

exit 0
```

Vedi Example 12-23, Example 3-4 ed Example 10-9 per script che utilizzano **find**. La relativa pagina di manuale fornisce tutti i dettagli di questo comando potente e complesso.

xargs

Un filtro per fornire argomenti ad un comando ed anche uno strumento per assemblare comandi. Suddivide il flusso di dati in parti sufficientemente piccole per essere elaborate da filtri o comandi. Lo si consideri un potente sostituto degli apici inversi. In situazioni in cui questi possono fallire con il messaggio d'errore "too many arguments", sostituendoli con **xargs**, spesso il problema si risolve. Normalmente **xargs** legge dallo `stdin` o da una pipe, ma anche dall'output di un file.

Il comando di default per **xargs** è echo. Questo significa che l'input collegato a **xargs** perde i ritorni a capo o qualsiasi altro carattere di spaziatura.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file2


bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file2
```

`ls | xargs -p -l gzip` comprime con gzip tutti i file della directory corrente, uno alla volta, ed attende un INVIO prima di ogni operazione.

> **Tip:** Un'interessante opzione di **xargs** è `-n NN`, che limita a *NN* il numero degli argomenti passati.
>
> **`ls | xargs -n 8 echo`** elenca i file della directory corrente su `8` colonne.

> **Tip:** Un'altra utile opzione è `-0`, in abbinamento con **find -print0** o **grep -lZ**. Permette di gestire gli argomenti che contengono spazi o apici.
>
> ```
> find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f
> grep -rliwZ GUI / | xargs -0 rm -f
> ```
>
> Entrambi gli esempi precedenti cancellano tutti i file che contengono "GUI". *(Grazie, S.C.)*

Example 12-4. Creare un file di log utilizzando xargs per verificare i log di sistema

```
#!/bin/bash

#  Genera un file di log nella directory corrente
#+ partendo dalla fine del file /var/log/messages.

#  Nota: /var/log/messages deve avere i permessi di lettura
#+ nel caso lo script venga invocato da un utente ordinario.
#         #root chmod 644 /var/log/messages

RIGHE=5

( date; uname -a ) >>logfile
# Data e nome della macchina
echo ------------------------------------------------------------ >>logfile
tail -$RIGHE /var/log/messages | xargs |  fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0

# Esercizio:
# ---------
#  Modificate lo script in modo che registri i cambiamenti avvenuti
#+ in /var/log/messages ad intervalli di venti minuti.
#  Suggerimento: usate il comando "watch".
```

Example 12-5. copydir, copiare i file della directory corrente in un’altra, utilizzando xargs

```
#!/bin/bash

#  Copia (con dettagli) tutti i file della directory corrente
#+ nella directory specificata da riga di comando.

E_NOARG=65

if [ -z "$1" ]   # Esce se non viene fornito nessun argomento.
then
  echo "Utilizzo: ‘basename $0‘ directory-in-cui-copiare"
  exit $E_NOARG
fi

ls . | xargs -i -t cp ./{} $1
# È l’equivalente esatto di
#    cp * $1
# a meno che qualche nome di file contenga caratteri di "spaziatura".

exit 0
```

Example 12-6. Analisi di frequenza delle parole utilizzando xargs

```
#!/bin/bash
# wf2.sh: Analisi sommaria della frequenza delle parole in un file di testo.

# Usa ’xargs’ per scomporre le righe del testo in parole singole.
# Confrontate quest’esempio con lo script "wf.sh" successivo.


# Verifica la presenza di un file di input passato da riga di comando.
ARG=1
E_ERR_ARG=65
E_NOFILE=66

if [ $# -ne "$ARG" ]
# Il numero di argomenti passati allo script è corretto?
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG
fi

if [ ! -f "$1" ]       # Verifica se il file esiste.
then
  echo "Il file \"$1\" non esiste."
  exit $E_NOFILE
fi



############################################################
cat "$1" | xargs -n1 | \
#  Elenca il file una parola per riga.
tr A-Z a-z | \
# Cambia tutte le lettere maiuscole in minuscole.
sed -e ’s/\.//g’  -e ’s/\,//g’ -e ’s/ /\
/g’ | \
#  Filtra i punti e le virgole, e
#+ cambia gli spazi tra le parole in linefeed.
  sort | uniq -c | sort -nr
#  Infine premette il conteggio delle occorrenze e le
#+ ordina in base al numero.
############################################################

#  Svolge lo stesso lavoro dell’esempio "wf.sh" che segue,
#+ ma in modo un po’ più greve e lento.

exit 0
```

`expr`

Comando multiuso per la valutazione delle espressioni: Concatena e valuta gli argomenti secondo le operazioni specificate (gli argomenti devono essere separati da spazi). Le operazioni possono essere aritmetiche, confronti, su stringhe o logiche.

`expr 3 + 5`

restituisce 8

`expr 5 % 3`

restituisce 2

`expr 5 \* 3`

restituisce 15

L'operatore di moltiplicazione deve essere usato con l'"escaping" nelle espressioni aritmetiche con **expr**.

`y=‘expr $y + 1‘`

Incrementa la variabile, con lo stesso risultato di **`let y=y+1`** e **`y=$(($y+1))`**. Questo è un esempio di espansione aritmetica.

`z=‘expr substr $stringa $posizione $lunghezza‘`

Estrae da $stringa una sottostringa di $lunghezza caratteri, iniziando da $posizione.

Example 12-7. Utilizzo di expr

```
#!/bin/bash

# Dimostrazione di alcuni degli usi di ’expr’
# ===========================================

echo

# Operatori aritmetici
# --------- ----------

echo "Operatori aritmetici"
echo
a=‘expr 5 + 3‘
echo "5 + 3 = $a"

a=‘expr $a + 1‘
echo
echo "a + 1 = $a"
echo "(incremento di variabile)"

a=‘expr 5 % 3‘
# modulo
echo
```

```
echo "5 modulo 3 = $a"

echo
echo

# Operatori logici
# --------- ------

#  Restituisce 1 per vero, 0 per falso,
#+ il contrario della normale convenzione Bash.

echo "Operatori logici"
echo

x=24
y=25
b=‘expr $x = $y‘         # Verifica l’uguaglianza.
echo "b = $b"            # 0  ( $x -ne $y )
echo

a=3
b=‘expr $a \> 10‘
echo ’b=‘expr $a \> 10‘, quindi...’
echo "Se a > 10, b = 0 ((falso)"
echo "b = $b"            # 0  ( 3 ! -gt 10 )
echo

b=‘expr $a \< 10‘
echo "Se a < 10, b = 1 (vero)"
echo "b = $b"            # 1  ( 3 -lt 10 )
echo
# Notate l’uso dell’escaping degli operatori.

b=‘expr $a \<= 3‘
echo "Se a <= 3, b = 1 (vero)"
echo "b = $b"            # 1  ( 3 -le 3 )
# Esiste anche l’operatore "\>=" (maggiore di o uguale a).


echo
echo

# Operatori di confronto
# --------- -- ---------

echo "Operatori di confronto"
echo
a=zipper
echo "a è $a"
if [ ‘expr $a = snap‘ ]
# Forza la ri-valutazione della variabile ’a’
then
   echo "a non è zipper"
```

```
fi

echo
echo



# Operatori per stringhe
# --------- --- --------

echo "Operatori per stringhe"
echo

a=1234zipper43231
echo "La stringa su cui opereremo è \"$a\"."

# length: lunghezza della stringa
b=‘expr length $a‘
echo "La lunghezza di \"$a\" è $b."

# index: posizione, in stringa, del primo carattere
#        della sottostringa verificato
b=‘expr index $a 23‘
echo "La posizione numerica del primo \"2\" in \"$a\" è \"$b\"."

#  substr: estrae una sottostringa, iniziando da posizione & lunghezza
#+ specificate
b=‘expr substr $a 2 6‘
echo "La sottostringa di \"$a\", iniziando dalla posizione 2,\
e con lunghezza 6 caratteri è \"$b\"."


#  Il comportamento preimpostato delle operazioni ’match’ è quello
#+ di cercare l’occorrenza specificata all’***inizio*** della stringa.
#
#        usa le Espressioni Regolari

b=‘expr match "$a" ’[0-9]*’‘     #  Conteggio numerico.
echo "Il numero di cifre all’inizio di \"$a\" è $b."
b=‘expr match "$a" ’\([0-9]*\)’‘ #  Notate che le parentesi con l’escape
#                   ==      ==   #+ consentono la verifica della sottostringa.
echo "Le cifre all’inizio di \"$a\" sono \"$b\"."

echo

exit 0
```

Important: L’operatore : può sostituire **match**. Per esempio, **b=‘expr $a : [0-9]*‘** è l’equivalente esatto di **b=‘expr match $a [0-9]*‘** del listato precedente.

```
#!/bin/bash

echo
```

```
echo "Operazioni sulle stringhe usando il costrutto \"expr \$stringa : \""
echo "================================================="
echo

a=1234zipper5FLIPPER43231

echo "La stringa su cui opereremo è \"‘expr "$a" : ’\(.*\)’‘\"."
# Operatore di raggruppamento parentesi con escape. ==  ==

#       *************************
#+       Le parentesi con l’escape
#+      verificano una sottostringa
#       *************************


#  Se non si esegue l’escaping delle parentesi...
#+ allora ’expr’ converte l’operando stringa in un intero.

echo "La lunghezza di \"$a\" è ‘expr "$a" : ’.*’‘."# Lunghezza della stringa

echo "Il numero di cifre all’inizio di \"$a\" è ‘expr "$a" : ’[0-9]*’‘."

# ------------------------------------------------------------------------ #

echo

echo "Le cifre all’inizio di \"$a\" sono ‘expr "$a" : ’\([0-9]*\)’‘."
#                                                  ==      ==
echo "I primi 7 caratteri di \"$a\" sono ‘expr "$a" : ’\(.......\)’‘."
#       =====                                      ==       ==
# Ancora, le parentesi con l’escape forzano la verifica della sottostringa.
#
echo"Gli ultimi 7 caratteri di \"$a\" sono ‘expr "$a" : ’.*\(.......\)’‘."
#        ======              operatore di fine stringa  ^^
#  (in realtà questo vuol dire saltare uno o più caratteri finché non viene
#+ raggiunta la sottostringa specificata)

echo

exit 0
```

Questo esempio illustra come **expr** usa le parentesi con *l’escape* -- *\(... \)* -- per raggruppare operatori, in coppia con la verifica di espressione regolare, per trovare una sottostringa.

Perl, sed e awk possiedono strumenti di gran lunga superiori per la verifica delle stringhe. Una breve “subroutine” **sed** o **awk** in uno script (vedi Section 34.2) è un’alternativa attraente all’uso di **expr**.

Vedi Section 9.2 per approfondimenti sulle operazioni su stringhe.

12.3. Comandi per ora/data

Ora/data e calcolo del tempo

date

La semplice invocazione di **date** visualizza la data e l'ora allo `stdout`. L'interesse per questo comando deriva dall'uso delle sue opzioni di formato e verifica.

Example 12-8. Utilizzo di date

```
#!/bin/bash
# Esercitarsi con il comando 'date'

echo "Il numero di giorni trascorsi dall'inizio dell'anno è `date +%j`."
# È necessario il '+' per il formato dell'output.
# %j fornisce i giorni dall'inizio dell'anno.

echo "Il numero di secondi trascorsi dal 01/01/1970 è `date +%s`."
#  %s contiene il numero di secondi dall'inizio della "UNIX epoch", ma
#+ quanto può essere utile?

prefisso=temp
suffisso=`eval date +%s`  # L'opzione "+%s" di 'date' è una specifica GNU.
nomefile=$prefisso.$suffisso
echo $nomefile
#  È importantissima per creare nomi di file temporanei "unici", è persino
#+ migliore dell'uso di $$.

# Leggete la pagina di manuale di 'date' per le altre opzioni di formato.

exit 0
```

L'opzione `-u` fornisce il tempo UTC (Universal Coordinated Time).

```
bash$ date
dom mag 11 17:55:55 CEST 2003



bash$ date -u
dom mag 11 15:56:08 UTC 2003
```

Il comando **date** possiede un certo numero di opzioni. Per esempio `%N` visualizza la parte di nanosecondi dell'ora corrente. Un uso interessante è quello per generare interi casuali di sei cifre.

```
date +%N | sed -e 's/000$//' -e 's/^0//'
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Toglie gli zeri iniziali e finali, se presenti.
```

Vedi anche Example 3-4.

zdump

Visualizza l'ora di una zona specifica.

```
bash$ zdump EST
EST  Sun May 11 11:01:53 2003 EST
```

time

Fornisce statistiche molto dettagliate sul tempo di esecuzione di un comando.

`time ls -l /` visualizza qualcosa di simile:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

Si veda anche il comando, molto simile, times nella sezione precedente.

> **Note:** Dalla versione 2.0 di Bash, **time** è diventata una parola riservata di shell, con un comportamento leggermente diverso se usato con una pipe.

touch

Utility per aggiornare all'ora corrente di sistema, o ad altra ora specificata, l'ora di accesso/modifica di un file. Viene usata anche per creare un nuovo file. Il comando **`touch zzz`** crea un nuovo file vuoto, di nome `zzz`, nell'ipotesi che `zzz` non sia già esistente. Creare file vuoti, che riportano l'ora e la data della loro creazione, può rappresentare un utile sistema per la registrazione del tempo, per esempio per tener traccia delle successive modifiche di un progetto.

> **Note:** Il comando **touch** equivale a : **`: >> nuovofile`** o a **`>> nuovofile`** (per i file regolari).

at

Il comando di controllo di job **at** esegue una data serie di comandi ad un'ora determinata. Ad uno sguardo superficiale, assomiglia a crond. Tuttavia, **at** viene utilizzato principalmente per eseguire la serie di comandi una sola volta.

`at 2pm January 15` visualizza un prompt per l'inserimento della serie di comandi da eseguire a quella data e ora. I comandi dovrebbero essere shell-script compatibili poiché, per questioni pratiche, l'utente sta digitando una riga alla volta in uno script di shell eseguibile. L'input deve terminare con un Ctl-D.

Con l'uso dell'opzione `-f` o della redirezione dell'input (<), **at** può leggere l'elenco dei comandi da un file. Questo file è uno script di shell eseguibile e, naturalmente, non dovrebbe essere interattivo. Risulta particolarmente intelligente inserire il comando run-parts nel file per eseguire una diversa serie di script.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

Il comando di controllo di job **batch** è simile ad **at**, ma esegue l'elenco dei comandi quando il carico di sistema cade sotto `0.8`. Come **at**, può leggere i comandi da un file con l'opzione `-f`.

cal

Visualizza allo `stdout` un calendario mensile in un formato molto elegante. Può fare riferimento all'anno corrente o ad un ampio intervallo di anni passati e futuri.

sleep

È l'equivalente shell di un ciclo wait. Sospende l'esecuzione per il numero di secondi indicato. È utile per la temporizzazione o per i processi in esecuzione in background che hanno il compito di verificare in continuazione il verificarsi di uno specifico evento (polling), come in Example 30-6.

```
sleep 3
# Pausa di 3 secondi.
```

> **Note:** Il comando **sleep** conta, in modo predefinito, i secondi. Possono però essere specificati minuti, ore o giorni.
>
> ```
> sleep 3 h
> # Pausa di 3 ore!
> ```

> **Note:** Per l'esecuzione di comandi da effettuarsi ad intervalli determinati, il comando watch può rivelarsi una scelta migliore di **sleep**.

usleep

Microsleep (la "u" deve interpretarsi come la lettera dell'alfabeto greco "mu", usata come prefisso per micro). È uguale a **sleep**, visto prima, ma "sospende" per intervalli di microsecondi. Può essere impiegato per una temporizzazione più accurata o per la verifica, ad intervalli di frequenza elevati, di un processo in esecuzione.

```
usleep 30
# Pausa di 30 microsecondi.
```

Questo comando fa parte del pacchetto Red Hat *initscripts / rc-scripts*.

> **Caution**
>
> Il comando **usleep** non esegue una temporizzazione particolarmente precisa e, quindi, non può essere impiegato per calcolare il tempo di cicli critici.

hwclock
clock

Il comando **hwclock** dà accesso e permette di regolare l'orologio hardware della macchina. Alcune opzioni richiedono i privilegi di root. Il file di avvio `/etc/rc.d/rc.sysinit` usa **hwclock** per impostare, in fase di boot, l'ora di sistema dall'orologio hardware.

Il comando **clock** è il sinonimo di **hwclock**.

12.4. Comandi di elaborazione testo

Comandi riguardanti il testo ed i file di testo

sort

Classificatore di file, spesso usato come filtro in una pipe. Questo comando ordina un flusso di testo, o un file, in senso crescente o decrescente, o secondo le diverse interpretazioni o posizioni dei caratteri. Usato con l'opzione `-m` unisce, in un unico file, i file di input precedentemente ordinati. La sua *pagina info* elenca le funzionalità e le molteplici opzioni di questo comando. Vedi Example 10-9, Example 10-10 e Example A-9.

tsort

Esegue un ordinamento topologico delle stringhe di un file di input lette in coppia.

uniq

Questo filtro elimina le righe duplicate di un file che è stato ordinato. È spesso usato in una pipe in coppia con sort.

```
cat lista-1 lista-2 lista-3 | sort | uniq > listafinale
# Vengono concatenati i file lista,
# ordinati,
# eliminate le righe doppie,
# ed infine il risultato è scritto in un file di output.
```

L'opzione `-c` premette ad ogni riga del file di input il numero delle sue occorrenze.

```
bash$ cat filetesto
```

```
Questa riga è presente una sola volta.
Questa riga è presente due volte.
Questa riga è presente due volte.
Questa riga è presente tre volte.
Questa riga è presente tre volte.
Questa riga è presente tre volte.


bash$ uniq -c filetesto
      1 Questa riga è presente una sola volta.
      2 Questa riga è presente due volte.
      3 Questa riga è presente tre volte.


bash$ sort filetesto | uniq -c | sort -nr
      3 Questa riga è presente tre volte.
      2 Questa riga è presente due volte.
      1 Questa riga è presente una sola volta.
```

La sequenza di comandi **`sort FILEINPUT | uniq -c | sort -nr`** produce un elenco delle *frequenze di occorrenza* riferite al file `FILEINPUT` (le opzioni `-nr` di **sort** generano un ordinamento numerico inverso). Questo modello viene usato nell'analisi dei file di log e nelle liste dizionario, od ogni volta che è necessario esaminare la struttura lessicale di un documento.

Example 12-9. Analisi delle frequenze delle parole

```
#!/bin/bash
#  wf.sh: Un'analisi sommaria, su un file di testo, della
#+ frequenza delle parole.
#  È una versione più efficiente dello script "wf2.sh".


# Verifica la presenza di un file di input passato da riga di comando.
ARG=1
E_ERR_ARG=65
E_NOFILE=66

if [ $# -ne "$ARG" ]  #  Il numero di argomenti passati allo script è corretto?
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG
fi

if [ ! -f "$1" ]      # Verifica se il file esiste.
then
  echo "Il file \"$1\" non esiste."
  exit $E_NOFILE
fi
```

```
###############################################################################
# main ()
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                           ========================
#                           Frequenza delle occorrenze

#  Filtra i punti e le virgole, e cambia gli spazi tra le parole in
#+ linefeed, quindi trasforma tutti i caratteri in caratteri minuscoli ed
#+ infine premette il conteggio delle occorrenze e le ordina in base al numero.
###############################################################################

# Esercizi:
# ---------
# 1)  Aggiungete dei comandi a 'sed' per filtrare altri segni di
#   + punteggiatura, come i punti e virgola.
# 2)  Modificatelo per filtrare anche gli spazi multipli e gli altri
#   + caratteri di spaziatura.
# 3)  Aggiungete una seconda chiave di ordinamento di modo che, nel
#   + caso di occorrenze uguali, queste vengano ordinate alfabeticamente.

exit 0


bash$ cat filetesto
 Questa riga è presente una sola volta.
 Questa riga è presente due volte.
 Questa riga è presente due volte.
 Questa riga è presente tre volte.
 Questa riga è presente tre volte.
 Questa riga è presente tre volte.


bash$ ./wf.sh filetesto
       6 riga
       6 questa
       6 presente
       6 è
       5 volte
       3 tre
       2 due
       1 volta
       1 una
       1 sola
```

expand
unexpand

Il filtro **expand** trasforma le tabulazioni in spazi. È spesso usato in una pipe.

Il filtro **unexpand** trasforma gli spazi in tabulazioni. Esegue l'azione opposta di **expand**.

cut

Strumento per estrarre i campi dai file. È simile alla serie di comandi **`print $N`** di awk, ma con capacità più limitate. In uno script è più semplice usare **cut** che non **awk**. Particolarmente importanti sono le opzioni `-d` (delimitatore) e `-f` (indicatore di campo - field specifier).

Usare **cut** per ottenere l'elenco dei filesystem montati:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

Usare **cut** per visualizzare la versione del SO e del kernel:

```
uname -a | cut -d" " -f1,3,11,12
```

Usare **cut** per estrarre le intestazioni dei messaggi da una cartella e-mail:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 Re: Spam complaint
```

Usare **cut** per la verifica di un file:

```
# Elenca tutti gli utenti presenti nel file /etc/passwd.

NOMEFILE=/etc/passwd

for utente in $(cut -d: -f1 $NOMEFILE)
do
  echo $utente
done

# Grazie, Oleg Philon per il suggerimento.
```

`cut -d ' ' -f2,3 nomefile` equivale a **`awk -F'[ ]' '{ print $2, $3 }' nomefile`**

Vedi anche Example 12-34.

paste

Strumento per unire più file in un unico file impaginato in diverse colonne. In combinazione con **cut** è utile per creare file di log di sistema.

join

Lo si può considerare il cugino specializzato di **paste**. Questa potente utility consente di fondere due file in modo da fornire un risultato estremamente interessante. Crea, in sostanza, una versione semplificata di un database relazionale.

Il comando **join** opera solo su due file, ma unisce soltanto quelle righe che possiedono una corrispondenza di campo comune (solitamente un'etichetta numerica) e visualizza il risultato allo `stdout`. I file che devono essere uniti devono essere anche ordinati in base al campo comune, se si vuole che l'abbinamento delle righe avvenga correttamente.

```
File: 1.dat

100 Scarpe
200 Lacci
300 Calze


File: 2.dat

100 EUR 40.00
200 EUR 1.00
300 EUR 2.00


bash$ join 1.dat 2.dat
File: 1.dat 2.dat

 100 Scarpe EUR 40.00
 200 Lacci  EUR 1.00
 300 Calze  EUR 2.00
```

Note: Il campo comune, nell'output, compare una sola volta.

head

visualizza la parte iniziale di un file allo `stdout` (il numero di righe preimpostato è `10`, il valore può essere modificato). Possiede un certo numero di opzioni interessanti.

Example 12-10. Quali file sono degli script?

```
#!/bin/bash
# script-detector.sh: Rileva gli script presenti in una directory.

VERCAR=2          # Verifica i primi 2 caratteri.
INTERPRETE='#!'   # Gli script iniziano con "#!".
```

```
for file in *     # Verifica tutti i file della directory corrente.
do
  if [[ ‘head -c$VERCAR "$file"‘ = "$INTERPRETE" ]]
  #      head -c2                   #!
  #  L’opzione ’-c’ di "head" agisce sul numero di caratteri specificato
  #+ anzichè sulle righe (comportamento di default).
  then
      echo "Il file \"$file\" è uno script."
  else
      echo "Il file \"$file\"  *non* è uno script."
  fi
done

exit 0
```

Example 12-11. Generare numeri casuali di 10 cifre

```
#!/bin/bash
# rnd.sh: Visualizza un numero casuale di 10 cifre

# Script di Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne ’1s/.* //p’


# ============================================================== #

# Analisi
# --------

# head:
# l’opzione -c4 prende i primi 4 byte.

# od:
# L’opzione -N4 limita l’output a 4 byte.
# L’opzione -tu4 seleziona il formato decimale senza segno per l’output.

# sed:
#  L’opzione -n, in combinazione con l’opzione "p" del comando "s", prende
#+ in considerazione, per l’output, solo le righe verificate.



# L’autore di questo script spiega l’azione di ’sed’ come segue.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne ’1s/.* //p’
# ---------------------------------> |

# Assumiamo che l’output fino a "sed">| sia 0000000 1198195154\n

#  sed inizia leggendo i caratteri: 0000000 1198195154\n.
#  Qui trova il carattere di ritorno a capo, quindi è pronto per elaborare
```

```
#+ la prima riga (0000000 1198195154), che assomiglia alla sua direttiva
#+ <righe><comandi>. La prima ed unica è

#   righe     comandi
#   1         s/.* //p

#  Il numero della riga è nell'intervallo, quindi entra in azione:
#+ cerca di sostituire la stringa più lunga terminante con uno spazio
#+ ("0000000 ") con niente "//" e in caso di successo, visualizza il risultato
#+ ("p" è l'opzione del comando "s", ed è differente dal comando "p").

#  sed ora continua la lettura dell'input. (Notate che prima di continuare, se
#+ non fosse stata passata l'opzione -n, sed avrebbe visualizzato la riga
#+ un'altra volta).

#  sed adesso legge la parte di caratteri rimanente, e trova la fine del file.
#  Si appresta ad elaborare la seconda riga (che può anche essere numerata
#+ con '$' perché è l'ultima).
#  Constata che non è compresa in <righe> e quindi termina il lavoro.

#  In poche parole, questo comando sed significa: "Solo sulla prima riga, togli
#+ qualsiasi carattere fino allo spazio, quindi visualizza il resto."

# Un modo migliore per ottenere lo stesso risultato sarebbe stato:
#           sed -e 's/.* //;q'

#  Qui abbiamo due <righe> e due <comandi> (si sarebbe potuto anche scrivere
#           sed -e 's/.* //' -e q):

#   righe                       comandi
#   niente (verifica la riga)   s/.* //
#   niente (verifica la riga)   q (quit)

#  In questo esempio, sed legge solo la sua prima riga di input.
#  Esegue entrambi i comandi e visualizza la riga (con la sostituzione) prima
#+ di uscire (a causa del comando "q"), perché non gli è stata passata
#+ l'opzione "-n".

# ================================================================== #

#  Un'alternativa più semplice al precedente script di una sola riga
#+ potrebbe essere:
#          head -c4 /dev/urandom| od -An -tu4

exit 0
```

Vedi anche Example 12-31.

tail

visualizza la parte finale di un file allo `stdout` (il valore preimpostato è di `10` righe). Viene comunemente usato per tenere traccia delle modifiche al file di log di sistema con l'uso dell'opzione `-f`, che permette l'aggiunta delle righe di output al file.

Example 12-12. Utilizzare tail per monitorare il log di sistema

```
#!/bin/bash

nomefile=sys.log

cat /dev/null > $nomefile; echo "Creazione / cancellazione del file."
#  Crea il file nel caso non esista, mentre lo svuota se è già stato creato.
#  vanno bene anche : > nomefile   e   > nomefile.

tail /var/log/messages > $nomefile
# /var/log/messages deve avere i permessi di lettura perché lo script funzioni.

echo "$nomefile contiene la parte finale del log di sistema."

exit 0
```

Vedi anche Example 12-4, Example 12-31 e Example 30-6.

grep

Strumento di ricerca multifunzione che fa uso delle espressioni regolari. In origine era un comando/filtro del venerabile editor **ed**. g/re/p, vale a dire, *global - regular expression - print*.

grep *`modello`* [*`file`*...]

Ricerca nel/nei file indicato/i l'occorrenza di *`modello`*, dove *`modello`* può essere o un testo letterale o un'espressione regolare.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

Se non vengono specificati i file, **grep** funziona come filtro sullo `stdout`, come in una pipe.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
901 pts/1    S      0:00 grep clock
```

L'opzione `-i` abilita una ricerca che non fa distinzione tra maiuscole e minuscole.

L'opzione `-w` verifica solo le parole complete.

L'opzione `-l` elenca solo i file in cui la ricerca ha avuto successo, ma non le righe verificate.

L'opzione `-r` (ricorsiva) ricerca i file nella directory di lavoro corrente e in tutte le sue sottodirectory.

L'opzione `-n` visualizza le righe verificate insieme al loro numero.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
6:The GPL governs the distribution of the Linux operating system.
```

L'opzione `-v` (o `--invert-match`) *scarta* le righe verificate.

```
grep modello1 *.txt | grep -v modello2

# Verifica tutte le righe dei file "*.txt" contenenti "modello1",
# ma ***non*** quelle contenenti "modello2".
```

L'opzione `-c` (`--count`) fornisce il numero delle occorrenze, ma non le visualizza.

```
grep -c txt *.sgml   # ((numero di occorrenze di "txt" nei file "*.sgml")


#   grep -cz .
#            ^ punto
# significa conteggio (-c) zero-diviso (-z) elementi da cercare "."
# cioè, quelli non vuoti (contenenti almeno 1 carattere).
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 4
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
# Per default, i caratteri di a capo (\n) separano gli elementi da cercare.

# Notate che l'opzione -z è specifica del "grep" di GNU.


# Grazie, S.C.
```

Quando viene invocato con più di un file, **grep** specifica qual'è il file contenente le occorrenze.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
osinfo.txt:The GPL governs the distribution of the Linux operating system.
misc.txt:The Linux operating system is steadily gaining in popularity.
```

> **Tip:** Per forzare **grep** a visualizzare il nome del file quando ne è presente soltanto uno, si deve indicare come secondo file `/dev/null`
>
> ```
> bash$ grep Linux osinfo.txt /dev/null
> ```

```
    osinfo.txt:This is a file containing information about Linux.
    osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

Se la ricerca ha avuto successo, **grep** restituisce come exit status 0. Questo lo rende utile per un costrutto di verifica in uno script, specialmente in abbinamento con l'opzione `-q` che sopprime l'output.

```
SUCCESSO=0                      # se la ricerca di grep è riuscita
parola=Linux
nomefile=file.dati

grep -q "$parola" "$nomefile"
# L'opzione "-q" non visualizza nulla allo stdout.

if [ $? -eq $SUCCESSO ]
then
  echo "$parola è presente in $nomefile"
else
  echo "$parola non è presente in $nomefile"
fi
```

L'Example 30-6 dimostra come usare **grep** per cercare una parola in un file di log di sistema.

Example 12-13. Simulare "grep" in uno script

```
#!/bin/bash
# grp.sh: Una reimplementazione molto sommaria di 'grep'.

E_ERR_ARG=65

if [ -z "$1" ]    # Verifica se sono stati passati argomenti allo script.
then
  echo "Utilizzo: 'basename $0' modello"
  exit $E_ERR_ARG
fi

echo

for file in *     # Verifica tutti i file in $PWD.
do
  output=$(sed -n /"$1"/p $file)  # Sostituzione di comando.

  if [ ! -z "$output" ]           #  Cosa succede se si usa "$output"
             #+ senza i doppi apici?
  then
      echo -n "$file: "
      echo $output
  fi              # sed -ne "/$1/s|^|${file}: |p" equivale al precedente.
```

```
  echo
done

echo

exit 0

# Esercizi:
# ---------
#  1) Aggiungete nuove righe di output nel caso ci sia più di una
#+    occorrenza per file.
# 2) Aggiungete altre funzionalità.
```

> **Note: egrep** è uguale a **grep -E**. Tuttavia usa una serie leggermente diversa ed estesa di espressioni regolari che possono rendere la ricerca un po' più flessibile.
>
> **fgrep** è uguale a **grep -F**. Esegue la ricerca letterale della stringa (niente espressioni regolari), il che velocizza sensibilmente l'operazione.
>
> **agrep** estende le capacità di **grep** per una ricerca per approssimazione. La stringa da ricercare differisce per un numero specifico di caratteri dalle occorrenze effettivamente risultanti. Questa utility non è, di norma, inclusa in una distribuzione Linux.

> **Tip:** Per la ricerca in file compressi vanno usati i comandi **zgrep**, **zegrep**, o **zfgrep**. Sebbene possano essere usati anche con i file non compressi, svolgono il loro compito più lentamente che non **grep**, **egrep**, **fgrep**. Sono invece utili per la ricerca in una serie di file misti, alcuni compressi altri no.
>
> Per la ricerca in file compressi con bzip si usa il comando **bzgrep**.

look

Il comando **look** opera come **grep**, ma la ricerca viene svolta in un "dizionario", un elenco di parole ordinate. In modo predefinito, **look** esegue la ricerca in `/usr/dict/words`. Naturalmente si può specificare un diverso percorso del file dizionario.

Example 12-14. Verificare la validità delle parole con un dizionario

```
#!/bin/bash
# lookup: Esegue una verifica di dizionario di tutte le parole di un file dati.

file=file.dati     # File dati le cui parole devono essere controllate.

echo

while [ "$Parola" != end ]  # Ultima parola del file dati.
do
  read parola      #  Dal file dati, a seguito della redirezione a fine ciclo.
  look $parola > /dev/null  #  Per non visualizzare le righe del
                            #+ file dizionario.
```

```
  verifica=$?      # Exit status del comando 'look'.

  if [ "$verifica" -eq 0 ]
  then
     echo "\"$parola\" è valida."
  else
     echo "\"$parola\" non è valida."
  fi

done <"$file"      #  Redirige lo stdin a $file, in modo che "read" agisca
                   #+ su questo.

echo

exit 0

# ----------------------------------------------------------------
#  Le righe di codice seguenti non vengono eseguite a causa del
#+ precedente comando "exit".


# Stephane Chazelas propone la seguente, e più concisa, alternativa:

while read parola && [[ $parola != end ]]
do if look "$parola" > /dev/null
   then echo "\"$parola\" è valida."
   else echo "\"$parola\" non è valida."
   fi
done <"$file"

exit 0
```

sed
awk

Linguaggi di scripting particolarmente adatti per la verifica di file di testo e dell'output dei comandi. Possono essere inseriti, singolarmente o abbinati, nelle pipe e negli script di shell.

sed

"Editor di flusso" non interattivo, consente l'utilizzo di molti comandi **ex** in modalità batch. Viene impiegato principalmente negli script di shell.

awk

Analizzatore e rielaboratore programmabile di file, ottimo per manipolare e/o localizzare campi (colonne) in file di testo strutturati. Ha una sintassi simile a quella del linguaggio C.

wc

wc fornisce il “numero di parole (‘word count’)” presenti in un file o in un flusso I/O:

```
bash $ wc /usr/doc/sed-3.02/README
20     127     838 /usr/doc/sed-3.02/README
[20 lines  127 words  838 characters]
```

wc -w fornisce solo il numero delle parole.

wc -l fornisce solo il numero di righe.

wc -c fornisce solo il numero di caratteri.

wc -L fornisce solo la dimensione della riga più lunga.

Uso di **wc** per contare quanti file *.txt* sono presenti nella directory di lavoro corrente:

```
$ ls *.txt | wc -l
#  Il conteggio si interrompe se viene trovato un carattere di
#+ linefeed nel nome di uno dei file "*.txt".

# Modi alternativi per svolgere lo stesso compito:
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)

# Grazie, S.C.
```

Uso di **wc** per calcolare la dimensione totale di tutti i file i cui nomi iniziano con le lettere comprese nell’intervallo d - h.

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Uso di **wc** per contare le occorrenze della parola “Linux” nel file sorgente di questo libro.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

Vedi anche Example 12-31 e Example 16-7.

Alcuni comandi possiedono, sotto forma di opzioni, alcune delle funzionalità di **wc**.

```
... | grep foo | wc -l
#  Questo costrutto, frequentemente usato, può essere reso in modo più conciso.

... | grep -c foo
# Un semplice impiego dell’opzione "-c" (o "--count") di grep.

# Grazie, S.C.
```

tr

filtro per la sostituzione di caratteri.

> **Caution**
>
> Si deve usare il “quoting” e/o le parentesi quadre, in modo appropriato. Il quoting evita la reinterpretazione dei caratteri speciali nelle sequenze di comandi **tr**. Va usato il quoting delle parentesi quadre se si vuole evitarne l’espansione da parte della shell.

Sia **`tr "A-Z" "*" <nomefile`** che **`tr A-Z \* <nomefile`** cambiano tutte le lettere maiuscole presenti in `nomefile` in asterischi (allo `stdout`). Su alcuni sistemi questo potrebbe non funzionare. A differenza di **`tr A-Z '[**]'`**.

L’opzione `-d` cancella un intervallo di caratteri.

```
echo "abcdef"                 # abcdef
echo "abcdef" | tr -d b-d     # aef


tr -d 0-9 <nomefile
# Cancella tutte le cifre dal file "nomefile".
```

L’opzione `--squeeze-repeats` (o `-s`) cancella tutte le occorrenze di una stringa di caratteri consecutivi, tranne la prima. È utile per togliere gli spazi in eccesso.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

L’opzione `-c` “complemento” *inverte* la serie di caratteri da verificare. Con questa opzione, **tr** agisce soltanto su quei caratteri che *non* verificano la serie specificata.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

È importante notare che **tr** riconosce le classi di caratteri POSIX. [2]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Example 12-15. toupper: Trasforma tutte le lettere di un file in maiuscole

```
#!/bin/bash
# Modifica tutte le lettere del file in maiuscole.

E_ERR_ARG=65

if [ -z "$1" ]  # Verifica standard degli argomenti da riga di comando.
```

```
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG
fi

tr a-z A-Z <"$1"

# Stesso effetto del precedente, ma usando la notazione dei caratteri POSIX:
#        tr ’[:lower:]’ ’[:upper:]’ <"$1"
# Grazie, S.C.


exit 0
```

Example 12-16. lowercase: Modifica tutti i nomi dei file della directory corrente in lettere minuscole

```
#! /bin/bash
#
#  Cambia ogni nome di file della directory di lavoro in lettere minuscole.
#
#  Ispirato da uno script di John Dubois, che è stato tradotto in Bash da Chet
#+ Ramey e semplificato considerevolmente da Mendel Cooper, autore del libro.


for file in *                   # Controlla tutti i file della directory.
do
   fnome=‘basename $file‘
   n=‘echo $fnome | tr A-Z a-z‘  #  Cambia il nome del file in tutte
     #+ lettere minuscole.
   if [ "$fnome" != "$n" ]       #  Rinomina solo quei file che non
     #+ sono già in minuscolo.
   then
      mv $fnome $n
   fi
done

exit 0


#  Il codice che si trova oltre questa riga non viene eseguito per il
#+ precedente "exit".
#-------------------------------------------------------------------#
# Se volete eseguirlo, cancellate o commentate le righe precedenti.

#  Lo script visto sopra non funziona con nomi di file conteneti spazi
#+ o ritorni a capo.

# Stephane Chazelas, quindi, suggerisce l’alternativa seguente:


for file in *        #  Non è necessario usare basename, perché "*" non
                     #+ restituisce i nomi di file contenenti "/".
```

```
do n=‘echo "$file/" | tr ’[:upper:]’ ’[:lower:]’‘

#                    Notazione POSIX dei set di caratteri.
#                    È stata aggiunta una barra, in modo che gli
#                    eventuali ritorni a capo non vengano cancellati
#                    dalla sostituzione di comando.
  # Sostituzione di variabile:
  n=${n%/}           #  Rimuove le barre, aggiunte precedentemente, dal
                     #+ nome del file.

  [[ $file == $n ]] || mv "$file" "$n"

     # Verifica se il nome del file è già in minuscolo.

done

exit 0
```

Example 12-17. Du: Conversione di file di testo DOS in formato UNIX

```
#!/bin/bash
# Du.sh: converte i file di testo DOS in formato UNIX .

E_ERR_ARG=65

if [ -z "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ nomefile-da-convertire"
  exit $E_ERR_ARG
fi

NUOVONOMEFILE=$1.unx

CR=’\015’  # Ritorno a capo.
           # 015 è il codice ottale ASCII di CR
           # Le righe dei file di testo DOS terminano con un CR-LF.

tr -d $CR < $1 > $NUOVONOMEFILE
# Cancella i CR e scrive il file nuovo.

echo "Il file di testo originale DOS è \"$1\"."
echo "Il file di testo tradotto in formato UNIX è \"$NOMENUOVOFILE\"."

exit 0

# Esercizio:
#-----------
# Modificate lo script per la conversione inversa (da UNIX a DOS).
```

Example 12-18. rot13: cifratura ultra-debole

```
#!/bin/bash
# rot13.sh: Classico algoritmo rot13, cifratura che potrebbe beffare solo un
#           bambino di 3 anni.

# Utilizzo: ./rot13.sh nomefile
# o      ./rot13.sh <nomefile
# o      ./rot13.sh e fornire l'input da tastiera (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'  # "a" corrisponde a "n", "b" a "o", ecc.
#  Il costrutto 'cat "$@"' consente di gestire un input proveniente sia dallo
#+ stdin che da un file.

exit 0
```

Example 12-19. Generare "Rompicapi Cifrati" di frasi celebri

```
#!/bin/bash
# crypto-quote.sh: Cifra citazioni

#  Cifra frasi famose mediante una semplice sostituzione monoalfabetica.
#  Il risultato è simile ai rompicapi "Crypto Quote" delle pagine Op Ed
#+ del Sunday.


chiave=ETAOINSHRDLUBCFGJMQPVWZYXK
# La "chiave" non è nient'altro che l'alfabeto rimescolato.
# Modificando la "chiave" cambia la cifratura.

# Il costrutto 'cat "$@"' permette l'input sia dallo stdin che dai file.
# Se si usa lo stdin, l'input va terminato con un Control-D.
# Altrimenti occorre specificare il nome del file come parametro da riga
# di comando.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$chiave"
#        |  in maiuscolo  |     cifra
# Funziona con frasi formate da lettere minuscole, maiuscole o entrambe.
# I caratteri non alfabetici non vengono modificati.


# Provate lo script con qualcosa di simile a
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# Il risultato è:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# Per decodificarlo:
# cat "$@" | tr "$chiave" "A-Z"
```

```
#  Questa semplice cifratura può essere spezzata da un dodicenne con il
#+ semplice uso di carta e penna.

exit 0
```

> **Le varianti di tr**
>
> L'utility **tr** ha due varianti storiche. La versione BSD che non usa le parentesi quadre (**`tr a-z A-Z`**), a differenza della versione SysV (**`tr '[a-z]' '[A-Z]'`**). La versione GNU di **tr** assomiglia a quella BSD, per cui è obbligatorio l'uso del quoting degli intervalli delle lettere all'interno delle parentesi quadre.

fold

Filtro che dimensiona le righe di input ad una larghezza specificata. È particolarmente utile con l'opzione `-s` che interrompe le righe in corrispondenza degli spazi tra una parola e l'altra (vedi Example 12-20 e Example A-2).

fmt

Semplice formattatore di file usato come filtro, in una pipe, per "ridimensionare" lunghe righe di testo per l'output.

Example 12-20. Elenco formattato di file

```
#!/bin/bash

AMPIEZZA=40                # Ampiezza di 40 colonne.

b=‘ls /usr/local/bin‘      # Esegue l’elenco dei file...

echo $b | fmt -w $AMPIEZZA

# Si sarebbe potuto fare anche con
#    echo $b | fold - -s -w $AMPIEZZA

exit 0
```

Vedi anche Example 12-4.

> **Tip:** Una potente alternativa a **fmt** è l'utility **par** di Kamil Toman, disponibile presso http://www.cs.berkeley.edu/~amc/Par/.

col

Questo filtro, dal nome fuorviante, rimuove i cosiddetti line feed inversi dal flusso di input. Cerca anche di sostituire gli spazi con caratteri di tabulazione. L'uso principale di **col** è quello di filtrare l'output proveniente da alcune utility di elaborazione di testo, come **groff** e **tbl**.

column

Riordina il testo in colonne. Questo filtro trasforma l'output di un testo, che apparirebbe come un elenco, in una "graziosa" tabella, inserendo caratteri di tabulazione in posizioni appropriate.

Example 12-21. Utilizzo di column per impaginare un elenco di directory

```
#!/bin/bash
#  L'esempio seguente corrisponde, con piccole modifiche, a quello
#+ contenuto nella pagina di manuale di "column".

(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t

#  "sed 1d" nella pipe cancella la prima riga di output, che sarebbe
#+ "total        N",
#+ dove "N" è il numero totale di file elencati da "ls -l".

# L'opzione -t di "column" visualizza l'output in forma tabellare.

exit 0
```

colrm

Filtro per la rimozione di colonne. Elimina le colonne (caratteri) da un file. Il risultato viene visualizzato allo `stdout`. **`colrm 2 4 <nomefile`** cancella dal secondo fino al quarto carattere di ogni riga del file di testo `nomefile`.

> **Warning**
>
> Se il file contiene caratteri non visualizzabili, o di tabulazione, il risultato potrebbe essere imprevedibile. In tali casi si consideri l'uso, in una pipe, dei comandi expand e **unexpand** posti prima di **colrm**.

nl

Filtro per l'enumerazione delle righe. **`nl nomefile`** visualizza nomefile allo `stdout` inserendo, all'inizio di ogni riga non vuota, il numero progressivo. Se nomefile viene omesso, l'azione viene svolta sullo `stdin`..

L'output di **nl** assomiglia molto a quello di **`cat -n`**, tuttavia, in modo predefinito, **nl** non visualizza le righe vuote.

Example 12-22. nl: Uno script che numera le proprie righe

```
#!/bin/bash

# Questo script si auto-visualizza due volte con le righe numerate.

#  'nl' considera questa riga come la nr. 3 perché le righe
#+ vuote vengono saltate.
# 'cat -n' vede la riga precedente come la numero 5.

nl 'basename $0'

echo; echo  # Ora proviamo con 'cat -n'

cat -n 'basename $0'
# La differenza è che 'cat -n' numera le righe vuote.
# Notate che lo stesso risultato lo si ottiene con  'nl -ba'.

exit 0
#-----------------------------------------------------------------
```

pr

Filtro di formato di visualizzazione. Impagina i file (o lo `stdout`) in sezioni adatte alla visualizzazione su schermo e per la stampa hard copy. Diverse opzioni consentono la gestione di righe e colonne come, tra l'altro, abbinare e numerare le righe, impostare i margini, aggiungere intestazioni ed unire file. Il comando **pr** riunisce molte delle funzionalità di **nl**, **paste**, **fold**, **column** e **expand**.

`pr -o 5 --width=65 fileZZZ | more` visualizza sullo schermo una piacevole impaginazione del contenuto del file `fileZZZ` con i margini impostati a 5 e 65.

L'opzione `-d` è particolarmente utile per forzare la doppia spaziatura (stesso effetto di **sed -G**).

gettext

Il pacchetto GNU **gettext** è una serie di utility per la localizzazione e traduzione dei messaggi di output dei programmi in lingue straniere. Originariamente progettato per i programmi in C, ora supporta diversi linguaggi di scripting e di programmazione.

Il *programma* **gettext** viene usato anche negli script di shell. Vedi la relativa *`pagina info`*.

msgfmt

Programma per generare cataloghi di messaggi in formato binario. Viene utilizzato per la localizzazione.

iconv

Utility per cambiare la codifica (set di caratteri) del/dei file. Utilizzato principalmente per la localizzazione.

recode

Va considerato come la versione più elaborata del precedente **iconv**. Questa versatile utility, usata per modificare la codifica di un file, non fa parte dell'installazione standard di Linux.

TeX
gs

TeX e **Postscript** sono linguaggi per la composizione di testo usati per preparare copie per la stampa o per la visualizzazione a video.

TeX è l'elaborato sistema di composizione di Donald Knuth. Conviene spesso scrivere uno script di shell contenente tutte le opzioni e gli argomenti che vanno passati ad uno di questi linguaggi.

Ghostscript (**gs**) è l'interprete Postscript rilasciato sotto licenza GPL.

groff
tbl
eqn

Un altro linguaggio di composizione e visualizzazione formattata di testo è **groff**. È la versione GNU, migliorata, dell'ormai venerabile pacchetto UNIX **roff/troff**. Le *pagine di manuale* utilizzano **groff** (vedi Example A-1).

L'utility per l'elaborazione delle tabelle **tbl** viene considerata come parte di **groff** perché la sua funzione è quella di trasformare le istruzioni per la composizione delle tabelle in comandi **groff**.

Anche l'utility per l'elaborazione di equazioni **eqn** fa parte di **groff** e il suo compito è quello di trasformare le istruzioni per la composizione delle equazioni in comandi **groff**.

lex
yacc

L'analizzatore lessicale **lex** genera programmi per la verifica d'occorrenza. Sui sistemi Linux è stato sostituito dal programma non proprietario **flex**.

L'utility **yacc** crea un analizzatore lessicale basato su una serie di specifiche. Sui sistemi Linux è stato sostituito dal non proprietario **bison**.

12.5. Comandi per file ed archiviazione

Archiviazione

tar

È l'utility standard di archiviazione UNIX. Dall'originale programma per il salvataggio su nastro (*Tape ARchiving*), si è trasformata in un pacchetto con funzionalità più generali che può gestire ogni genere di archiviazione con qualsiasi tipo di dispositivo di destinazione, dai dispositivi a nastro ai file regolari fino allo `stdout` (vedi Example 3-4). Tar GNU è stato implementato per accettare vari filtri di compressione, ad esempio

tar czvf nome_archivio.tar.gz * che archivia ricorsivamente e comprime con gzip tutti i file, tranne quelli il cui nome inizia con un punto (dotfile), della directory di lavoro corrente (**$PWD**). [3]

Alcune utili opzioni di **tar**:

1. `-c` crea (un nuovo archivio)
2. `-x` estrae (file da un archivio esistente)
3. `--delete` cancella (file da un archivio esistente)

> **Caution**
>
> Questa opzione non funziona sui dispositivi a nastro magnetico.

4. `-r` accoda (file ad un archivio esistente)
5. `-A` accoda (file *tar* ad un archivio esistente)
6. `-t` elenca (il contenuto di un archivio esistente)
7. `-u` aggiorna l'archivio
8. `-d` confronta l'archivio con un filesystem specificato
9. `-z` usa gzip sull'archivio (lo comprime o lo decomprime in base all'abbinamento con l'opzione `-c` o `-x`)
10. `-j` usa bzip2 sull'archivio

> **Caution**
>
> Poiché potrebbe essere difficile ripristinare dati da un archivio tar compresso con *gzip* è consigliabile, per l'archiviazione di file importanti, eseguire salvataggi (backup) multipli.

shar

Utility di archiviazione di shell. I file di un archivio shell vengono concatenati senza compressione. Quello che risulta è essenzialmente uno script di shell, completo di intestazione #!/bin/sh e contenente tutti i necessari comandi di ripristino. Gli archivi shar fanno ancora la loro comparsa solo nei newsgroup Internet, dal momento che **shar** è stato sostituito molto bene da **tar**/**gzip**. Il comando **unshar** ripristina gli archivi **shar**.

ar

Utility per la creazione e la manipolazione di archivi, usata principalmente per le librerie di file oggetto binari.

rpm

Il *Red Hat Package Manager*, o utility **rpm**, è un gestore per archivi binari o sorgenti. Tra le altre cose, comprende comandi per l'installazione e la verifica dell'integrità dei pacchetti.

Un semplice **rpm -i nome_pacchetto.rpm** è di solito sufficiente per installare un pacchetto, sebbene siano disponibili molte più opzioni.

Tip: `rpm -qa` fornisce l'elenco completo dei pacchetti *rpm* installati su un sistema. **`rpm -qa nome_pacchetto`** elenca solo il pacchetto corrispondente a `nome_pacchetto`.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
glibc-2.2.4-13
cracklib-2.7-12
dosfstools-2.7-1
gdbm-1.8.0-10
ksymoops-2.4.1-1
mktemp-1.5-11
perl-5.6.0-17
reiserfs-utils-3.x.0j-2
...


bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2


bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
docbook-style-dsssl-1.64-3
docbook-dtd30-sgml-1.0-10
docbook-dtd40-sgml-1.0-11
docbook-utils-pdf-0.6.9-2
docbook-dtd41-sgml-1.0-10
docbook-utils-0.6.9-2
```

cpio

Comando specializzato per la copia di archivi (**c**o**p**y **i**nput and **o**utput), si incontra molto raramente, essendo stato soppiantato da **tar**/**gzip**. Le sue funzionalità, comunque, vengono ancora utilizzate, ad esempio per spostare una directory.

Example 12-23. Utilizzo di cpio per spostare una directory

```
#!/bin/bash

# Copiare una directory usando ’cpio.’

ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ]
then
  echo "Utilizzo: ‘basename $0‘ directory destinazione"
  exit $E_ERR_ARG
fi
```

```
directory=$1
destinazione=$2

find "$directory" -depth | cpio -admvp "$destinazione"
#                  ^^^^^         ^^^^^
# Leggete le pagine di manuale di 'find' e 'cpio' per decifrare queste opzioni.

#  Qui si potrebbe inserire il controllo dell'exit status ($?)
#+ per vedere se tutto ha funzionato correttamente.

exit 0
```

rpm2cpio

Questo comando estrae un archivio **cpio** da un archivio rpm.

Example 12-24. Decomprimere un archivio *rpm*

```
#!/bin/bash
# de-rpm.sh: Decomprime un archivio 'rpm'

: ${1?"Utilizzo: `basename $0` file_archivio"}
# Bisogna specificare come argomento un archivio 'rpm'.


TEMPFILE=$$.cpio              # File temporaneo con nome "unico".
                              # $$ è l'ID di processo dello script.

rpm2cpio < $1 > $TEMPFILE     # Converte l'archivio rpm in un archivio cpio.
cpio --make-directories -F $TEMPFILE -i  # Decomprime l'archivio cpio.
rm -f $TEMPFILE                          # Cancella l'archivio cpio.

exit 0

#  Esercizio:
#  Aggiungete dei controlli per verificare se
#+                       1) "file_archivio" esiste e
#+                       2) è veramente un archivio rpm.
#  Suggerimento:  verificate l'output del comando 'file'.
```

Compressione

gzip

Utility di compressione standard GNU/UNIX che ha sostituito la meno potente e proprietaria **compress**. Il corrispondente comando di decompressione è **gunzip**, equivalente a **gzip -d**.

Il filtro **zcat** decomprime un file compresso con *gzip* allo `stdout`, come input per una pipe o una redirezione. In effetti, è il comando **cat** che agisce sui file compressi (compresi quelli ottenuti con la vecchia utility **compress**). Il comando **zcat** equivale a **gzip -dc**.

> **Caution**
>
> Su alcuni sistemi commerciali UNIX, **zcat** è un sinonimo di **uncompress -c**, di conseguenza non funziona su file compressi con *gzip*.

Vedi anche Example 7-7.

bzip2

Utility di compressione alternativa, più efficiente (ma più lenta) di **gzip**, specialmente con file di ampie dimensioni. Il corrispondente comando di decompressione è **bunzip2**.

> **Note:** Le versioni più recenti di tar sono state aggiornate per supportare **bzip2**.

compress
uncompress

E' la vecchia utility proprietaria di compressione presente nelle distribuzioni commerciali UNIX. È stata ampiamente sostituita dalla più efficiente **gzip** . Le distribuzioni Linux includono, di solito, **compress** per ragioni di compatibilità, sebbene **gunzip** possa decomprimere i file trattati con **compress**.

> **Tip:** Il comando **znew** ricomprime i file dal formato *compress* al formato *gzip*.

sq

Altra utility di compressione. È un filtro che opera solo su elenchi di parole ASCII ordinate. Usa la sintassi standard dei filtri, **sq < file-input > file-output**. Veloce, ma non così efficiente come gzip. Il corrispondente filtro di decompressione è **unsq**, con la stessa sintassi di **sq**.

> **Tip:** L'output di **sq** può essere collegato per mezzo di una pipe a **gzip** per una ulteriore compressione.

zip
unzip

Utility di archiviazione e compressione multipiattaforma, compatibile con il programma DOS *pkzip.exe*. Gli archivi "zippati" sembrano rappresentare, su Internet, il mezzo di scambio più gradito rispetto ai "tarball".

unarc
unarj
unrar

Queste utility Linux consentono di decomprimere archivi compressi con i programmi DOS *arc.exe*, *arj.exe* e *rar.exe*.

Informazioni sui file

file

Utility per identificare i tipi di file. Il comando **`file nome_file`** restituisce la specifica di `nome_file`, come `ascii text` o `data`. Fa riferimento ai magic number che si trovano in `/usr/share/magic`, `/etc/magic` o `/usr/lib/magic`, secondo le distribuzioni Linux/UNIX.

L'opzione `-f` esegue **file** in modalità batch, per leggere l'elenco dei file contenuto nel file indicato. L'opzione `-z` tenta di verificare il formato e le caratteristiche dei file compressi.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified:
Sun Sep 16 13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last modified: Sun Sep 16 13:34:51 2001, os:
```

Example 12-25. Togliere i commenti da file C

```
#!/bin/bash
# strip-comment.sh: Toglie i commenti (/* COMMENTO */) in un programma C.

E_NOARG=65
E_ERR_ARG=66
E_TIPO_FILE_ERRATO=67

if [ $# -eq "$E_NOARG" ]
then
  echo "Utilizzo: ‘basename $0‘ file-C" >&2 #  Messaggio d’errore allo stderr.
  exit $E_ERR_ARG
fi

# Verifica il corretto tipo di file.
tipo=‘eval file $1 | awk ’{ print $2, $3, $4, $5 }’‘
# "file $1" restituisce nome e tipo di file . . .
# quindi awk rimuove il primo campo, il nome . . .
# Dopo di che il risultato è posto nella variabile "tipo".
tipo_corretto="ASCII C program text"

if [ "$tipo" != "$tipo_corretto" ]
then
  echo
  echo "Questo script funziona solo su file sorgenti C."
```

```
  echo
  exit $E_TIPO_FILE_ERRATO
fi


# Script sed piuttosto criptico:
#--------
sed ’
/^\/\*/d
/.*\/\*/d
’ $1
#--------
# Facile da capire, se dedicate diverse ore ad imparare i fondamenti di sed.


#  È necessario aggiungere ancora una riga allo script sed per trattare
#+ quei casi in cui una riga di codice è seguita da un commento.
#  Questo viene lasciato come esercizio (niente affatto banale).

#  Ancora, il codice precedente cancella le righe con un "*/" o "/*", che non
#+ è un risultato desiderabile.

exit 0


# ---------------------------------------------------------------------------
#  Il codice oltre la linea non viene eseguito a causa del precedente ’exit 0’.

# Stephane Chazelas suggerisce la seguente alternativa:

utilizzo() {
  echo "Utilizzo: ‘basename $0‘ file-C" >&2
  exit 1
}

STRANO=‘echo -n -e ’\377’‘   # oppure STRANO=$’\377’
[[ $# -eq 1 ]] || utilizzo
case ‘file "$1"‘ in
  *"C program text"*) sed -e "s%/\*%${STRANO}%g;s%\*/%${STRANO}%g" "$1" \
  | tr ’\377\n’ ’\n\377’ \
  | sed -ne ’p;n’ \
  | tr -d ’\n’ | tr ’\377’ ’\n’;;
  *) utilizzo;;
esac

#  Questo può ancora essere ingannato da occorrenze come:
#  printf("/*");
#  o
#  /*  /* errato commento annidato */
#
#  Per poter gestire tutti i casi particolari (commenti in stringhe, commenti
#+ in una stringa in cui è presente \", \\" ...) l’unico modo è scrivere un
#+ parser C (forse lex o yacc?).
```

```
exit 0
```

which

which comando-xxx restituisce il percorso di “comando-xxx”. È utile per verificare se un particolare comando o utility è installato sul sistema.

```
$bash which rm

/usr/bin/rm
```

whereis

Simile al precedente **which**, **whereis comando-xxx** restituisce il percorso di “comando-xxx” ed anche della sua *pagina di manuale*.

```
$bash whereis rm

rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis filexxx ricerca “filexxx” nel database *`whatis`*. È utile per identificare i comandi di sistema e i file di configurazione. Può essere considerato una semplificazione del comando **man**.

```
$bash whatis whatis

whatis              (1)  - search the whatis database for complete words
```

Example 12-26. Esplorare `/usr/X11R6/bin`

```
#!/bin/bash

# Cosa sono tutti quei misteriosi eseguibili in /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Provate anche "/bin", "/usr/bin", "/usr/local/bin", ecc.

for file in $DIRECTORY/*
do
  whatis ‘basename $file‘ # Visualizza le informazione sugli eseguibili.
done

exit 0
# Potreste desiderare di redirigere l’output di questo script, così:
# ./what.sh >>whatis.db
```

```
# o visualizzarne una pagina alla volta allo stdout,
# ./what.sh | less
```

Vedi anche Example 10-3.

vdir

Visualizza l'elenco dettagliato delle directory. L'effetto è simile a ls -l.

Questa è una delle *fileutils* GNU.

```
bash$ vdir
total 10
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate
slocate

Il comando **locate** esegue la ricerca dei file usando un database apposito. Il comando **slocate** è la versione di sicurezza di **locate** (che può essere l'alias di **slocate**).

```
$bash locate hickson

/usr/lib/xephem/catalogs/hickson.edb
```

readlink

Rivela il file a cui punta un link simbolico.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

Il comando **strings** viene usato per cercare le stringhe visualizzabili in un file dati o binario. Elenca le sequenze di caratteri trovate nel file di riferimento. E' utile per un esame rapido e sommario di un file core di scarico della memoria o per dare un'occhiata ad un file di immagine sconosciuto (**`strings file-immagine | more`** potrebbe restituire qualcosa come `JFIF` che indica un file grafico *jpeg*). In uno script, si può controllare l'output di **strings** con grep o sed. Vedi Example 10-7 e Example 10-9.

Example 12-27. Un comando *strings* "migliorato"

```
#!/bin/bash
# wstrings.sh: "word-strings" (comando "strings" migliorato)
#
#  Questo script filtra l'output di "strings" confrontandolo
#+ con un file dizionario.
#  In questo modo viene eliminato efficacemente il superfluo,
#+ restituendo solamente le parole riconosciute.

# =================================================================
#       Verifica Standard del/degli Argomento/i dello Script
ARG=1
E_ERR_ARG=65
E_NOFILE=66

if [ $# -ne $ARG ]
then
  echo "Utilizzo: `basename $0` nomefile"
  exit $E_ERR_ARG
fi

if [ ! -f "$1" ]                       # Verifica l'esistenza del file.
then
    echo "Il file \"$1\" non esiste."
    exit $E_NOFILE
fi
# =================================================================


LUNMINSTR=3                            #  Lunghezza minima della stringa.
DIZIONARIO=/usr/share/dict/linux.words #  File dizionario.
                                       #  Può essere specificato un file
                                       #+ dizionario diverso purché
                                       #+ di una parola per riga.


elenco=`strings "$nome_file" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' ''

#  Modifica l'output del comando 'strings' mediante diversi passaggi a 'tr'.
#  "tr A-Z a-z"  trasforma le lettere maiuscole in minuscole.
#  "tr '[:space:]' Z"  trasforma gli spazi in Z.
#  "tr -cs '[:alpha:]' Z"  trasforma i caratteri non alfabetici in Z,
#+ riducendo ad una sola le Z multiple consecutive.
```

```
#  "tr -s '\173-\377' Z"  trasforma tutti i caratteri oltre la 'z' in Z,
#+ riducendo ad una sola le Z multiple consecutive, liberandoci così di tutti i
#+ caratteri strani che la precedente istruzione non è riuscita a trattare.
#  Infine, "tr Z ' '" trasforma tutte queste Z in spazi, che saranno
#+ considerati separatori di parole dal ciclo che segue.

#  Notate la tecnica di concatenare diversi 'tr',
#+ ma con argomenti e/o opzioni differenti ad ogni passaggio.


for parola in $elenco                 #  Importante:
                                      #  non bisogna usare $elenco col quoting.
                                      #  "$elenco" non funziona.
                                      #  Perché?
do

  lunstr=${#parola}                   #  Lunghezza della stringa.
  if [ "$lunstr" -lt "$LUNMINSTR" ]   #  Salta le stringhe con meno
                                      #+ di 3 caratteri.
  then
    continue
  fi

  grep -Fw $parola "$DIZIONARIO"      #  Cerca solo le parole complete.
       ^^^                            #  Opzioni "stringhe Fisse" e
                                      #+ "parole (words) intere".

done


exit 0
```

Confronti

diff
patch

diff: flessibile utility per il confronto di file. Confronta i file di riferimento riga per riga, sequenzialmente. In alcune applicazioni, come nei confronti di dizionari, è vantaggioso filtrare i file di riferimento con sort e **uniq** prima di collegarli tramite una pipe a **diff**. **`diff file-1 file-2`** visualizza le righe dei file che differiscono, con le parentesi acute ad indicare a quale file ogni particolare riga appartiene.

L'opzione `--side-by-side` di **diff** visualizza riga per riga, in colonne separate, ogni file confrontato con un segno indicante le righe non coincidenti. Le opzioni `-c` e `-u`, similmente, rendono più facile l'interpretazione dell'output del comando.

Sono disponibili diversi front-end per **diff**, quali **spiff**, **wdiff**, **xdiff** e **mgdiff**.

> **Tip:** Il comando **diff** restituisce exit status 0 se i file confrontati sono identici, 1 in caso contrario. Questo consente di utilizzare **diff** per un costrutto di verifica in uno script di shell (vedi oltre).

L'uso più comune di **diff** è quello per creare file di differenze da utilizzare con **patch**. L'opzione `-e` produce script idonei all'utilizzo con l'editor **ed** o **ex**.

patch: flessibile utility per gli aggiornamenti. Dato un file di differenze prodotto da **diff**, **patch** riesce ad aggiornare un pacchetto alla versione più recente. È molto più conveniente distribuire un file di "differenze", di dimensioni relativamente minori, che non l'intero pacchetto aggiornato. Il "patching" del kernel è diventato il metodo preferito per la distribuzione delle frequenti release del kernel Linux.

```
patch -p1 <file-patch
# Prende tutte le modifiche elencate in 'file-patch'
# e le applica ai file che sono specificati in "file-patch".
# Questo esegue l'aggiornamento del pacchetto alla versione più recente.
```

Patch del kernel:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Aggiornamento dei sorgenti del kernel usando 'patch'.
# Dal file "README" della documentazione del kernel Linux,
# di autore anonimo (Alan Cox?).
```

> **Note:** Il comando **diff** riesce anche ad eseguire un confronto ricorsivo tra directory (sui file in esse contenuti).
>
> ```
> bash$ diff -r ~/notes1 ~/notes2
> Only in /home/bozo/notes1: file02
> Only in /home/bozo/notes1: file03
> Only in /home/bozo/notes2: file04
> ```

> **Tip:** Si usa **zdiff** per confrontare file compressi con *gzip*.

diff3

Versione estesa di **diff** che confronta tre file alla volta. Questo comando restituisce, come exit status, 0 in caso di successo, ma sfortunatamente non fornisce alcuna informazione sui risultati del confronto.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
    Questa è la riga 1 di "file-1".
 2:1c
    Questa è la riga 1 di "file-2".
```

```
 3:1c
    Questa è la riga 1 di "file-3"
```

sdiff

Confronta e/o visualizza due file con lo scopo di unirli in un unico file. A causa della sua natura interattiva, questo comando viene usato poco negli script.

cmp

Il comando **cmp** è la versione più semplice di **diff**. Mentre **diff** elenca le differenze tra i due file, **cmp** mostra semplicemente i punti in cui differiscono.

> **Note:** Come **diff**, **cmp** restituisce exit status 0 se i file confrontati sono identici, 1 in caso contrario. Questo ne consente l'impiego per un costrutto di verifica in uno script di shell.

Example 12-28. Utilizzare cmp in uno script per confrontare due file

```
#!/bin/bash

ARG=2  # Lo script si aspetta due argomenti.
E_ERR_ARG=65
E_NONLEGGIBILE=66

if [ $# -ne "$ARG" ]
then
  echo "Utilizzo: ‘basename $0‘ file1 file2"
  exit $E_ERR_ARG
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Entrambi i file, per essere confrontati, devono esistere ed avere"
  echo "i permessi di lettura."
  exit $E_NONLEGGIBILE
fi

cmp $1 $2 &> /dev/null  #  /dev/null elimina la visualizzazione del
                        #+ risultato del comando"cmp".
#   cmp -s $1 $2  ottiene lo stesso risultato (opzione "-s" di "cmp")
#   Grazie  Anders Gustavsson per averlo evidenziato.
#
# Funziona anche con ’diff’, vale a dire, diff $1 $2 &> /dev/null

if [ $? -eq 0 ]         # Verifica l’exit status del comando "cmp".
then
  echo "Il file \"$1\" è identico al file \"$2\"."
```

```
else
  echo "Il file \"$1\" è diverso dal file \"$2\"."
fi

exit 0
```

> **Tip:** Si usa **zcmp** per i file compressi con *gzip*.

comm

Versatile utility per il confronto di file. I file da confrontare devono essere ordinati.

comm *-opzioni primo-file secondo-file*

`comm file-1 file-2` visualizza il risultato su tre colonne:

- colonna 1 = righe uniche appartenenti a `file-1`
- colonna 2 = righe uniche appartenenti a `file-2`
- colonna 3 = righe comuni ad entrambi i file.

Alcune opzioni consentono la soppressione di una o più colonne di output.

- `-1` sopprime la colonna `1`
- `-2` sopprime la colonna `2`
- `-3` sopprime la colonna `3`
- `-12` sopprime entrambe le colonne `1` e `2`, etc.

Utility

basename

Elimina il percorso del file, visualizzando solamente il suo nome. Il costrutto **`basename $0`** permette allo script di conoscere il proprio nome, vale a dire, il nome con cui è stato invocato. Si può usare per i messaggi di "utilizzo" se, per esempio, uno script viene eseguito senza argomenti:

```
echo "Utilizzo: ‘basename $0‘ arg1 arg2 ... argn"
```

dirname

Elimina **basename**, dal nome del file, visualizzando solamente il suo percorso.

> **Note: basename** e **dirname** possono operare su una stringa qualsiasi. Non è necessario che l'argomento si riferisca ad un file esistente e neanche essere un nome di file (vedi Example A-8).

Example 12-29. basename e dirname

```
#!/bin/bash

a=/home/bozo/daily-journal.txt

echo "Basename di /home/bozo/daily-journal.txt = ‘basename $a‘"
echo "Dirname di /home/bozo/daily-journal.txt = ‘dirname $a‘"
echo
# Funzionano entrambe anche con la sola ~.
echo "La mia cartella personale è ‘basename ~/‘."
echo "La directory home della mia cartella personale è ‘dirname ~/‘."

exit 0
```

split

Utility per suddividere un file in porzioni di dimensioni minori. È solitamente impiegata per suddividere file di grandi dimensioni allo scopo di eseguirne un salvataggio su floppy disk, per l’invio tramite e-mail o per effettuarne l’upload su un server.

sum
cksum
md5sum

Sono utility per creare le checksum. Una *checksum* è un numero ricavato con un calcolo matematico eseguito sul contenuto di un file, con lo scopo di verificarne l’integrità. Uno script potrebbe verificare un elenco di checksum a fini di sicurezza, per esempio per assicurarsi che il contenuto di indispensabili file di sistema non sia stato modificato o corrotto. Per applicazioni di sicurezza, si dovrebbe utilizzare il comando **md5sum** a 128-bit (**m**essage **d**igest check**sum**).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz


bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba /boot/vmlinuz
```

E’ da notare che **cksum** visualizza anche la dimensione, in byte, del file di riferimento.

Example 12-30. Verificare l’integrità dei file

```
#!/bin/bash
# file-integrity.sh: Verifica se i file di una data directory
#                    sono stati modificati senza autorizzazione.
```

```
E_DIR_ERRATA=70
E_ERR_DBFILE=71

dbfile=File_record.md5
# Nome del file che contiene le checksum.


crea_database ()
{
  echo ""$directory"" > "$dbfile"
  # Scrive il nome della directory come prima riga di dbfile.
  md5sum "$directory"/* >> "$dbfile"
  # Accoda le checksum md5 e i nomi dei file.
}

verifica_database ()
{
  local n=0
  local nomefile
  local checksum

  # ------------------------------------------------------------ #
  #  Questa verifica potrebbe anche non essere necessaria, ma è
  #+ meglio essere pignoli che rischiare.

  if [ ! -r "$dbfile" ]
  then
      echo "Impossibile leggere il database delle checksum!"
      exit $E_ERR_DBFILE
  fi
  # ------------------------------------------------------------ #

  while read record[n]
  do

      directory_verificata="${record[0]}"
      if [ "$directory_verificata" != "$directory" ]
      then
            echo "Le directory non corrispondono!"
            # E' stato indicato un nome di directory sbagliato.
            exit $E_DIR_ERRATA
      fi

      if [ "$n" -gt 0 ]   # Non è il nome della directory.
      then
            nomefile[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
            #  md5sum scrive nel primo campo la checksum, nel
            #+ secondo il nome del file.
            checksum[n]=$( md5sum "${nomefile[n]}" )

            if [ "${record[n]}" = "${checksum[n]}" ]
            then
                    echo "${nomefile[n]} non è stato modificato."
```

```
            else
                    echo "${nomefile[n]} : CHECKSUM ERRATA!"
                    # Il file è stato modificato dall'ultima verifica.
            fi

      fi


      let "n+=1"
  done <"$dbfile"   # Legge il database delle checksum.

}

# ================================================== #
# main ()

if [ -z  "$1" ]
then
  directory="$PWD"      #  Se non altrimenti specificata, usa la
else                    #+ directory di lavoro corrente.
  directory="$1"
fi

clear                   # Pulisce lo schermo.

# ------------------------------------------------------------------- #
  if [ ! -r "$dbfile" ] # Occorre creare il database?
  then
      echo "Sto creando il database, \""$directory"/"$dbfile"\"."; echo
      crea_database
  fi
# ------------------------------------------------------------------- #

verifica_database       # Esegue il lavoro di verifica.

echo

#  Sarebbe desiderabile redirigere lo stdout dello script in un file,
#+ specialmente se la directory da verificare contiene molti file.

#  Per una verifica d'integrità molto più approfondita, considerate l'impiego
#+ del pacchetto "Tripwire", http://sourceforge.net/projects/tripwire/.

exit 0
```

Vedi anche Example A-20 per un uso creativo del comando **md5sum**.

shred

Cancella in modo sicuro un file, sovrascrivendolo diverse volte con caratteri casuali prima di cancellarlo definitivamente. Questo comando ha lo stesso effetto di Example 12-43, ma esegue il compito in una maniera più completa ed elegante.

Questa è una delle *fileutils* GNU.

> **Caution**
>
> Tecniche di indagine avanzate potrebbero essere ancora in grado di recuperare il contenuto di un file anche dopo l'uso di **shred**.

Codifica e Cifratura

uuencode

Questa utility codifica i file binari in caratteri ASCII, rendendoli disponibili per la trasmissione nel corpo di un messaggio e-mail o in un post di newsgroup.

uudecode

Inverte la codifica, ripristinando i file binari codificati con uuencode al loro stato originario.

Example 12-31. Decodificare file

```
#!/bin/bash

righe=35        # 35 righe di intestazione (molto generoso).

for File in *   # Verifica tutti i file nella directory di lavoro corrente...
do
  ricerca1=‘head -$righe $File | grep begin | wc -w‘
  ricerca2=‘tail -$righe $File | grep end | wc -w‘
  #  Decodifica i file che hanno un "begin" nella parte iniziale e un "end"
  #+ in quella finale.
  if [ "$ricerca1" -gt 0 ]
  then
      if [ "$ricerca2" -gt 0 ]
      then
            echo "Decodifico con uudecode - $File -"
            uudecode $File
    fi
  fi
done

#  Notate che se si invoca questo script su se stesso, l’esecuzione è ingannata
#+ perché pensa di trovarsi in presenza di un file codificato con uuencode,
#+ poiché contiene sia "begin" che "end".

# Esercizio:
# Modificate lo script per verificare l’intestazione di un newsgroup.
```

```
exit 0
```

> **Tip:** Il comando fold -s può essere utile (possibilmente in una pipe) per elaborare messaggi di testo di grandi dimensioni, decodificati con uudecode, scaricati dai newsgroup Usenet.

mimencode
mmencode

I comandi **mimencode** e **mmencode** elaborano gli allegati e-mail nei formati di codifica MIME. Sebbene i gestori di e-mail (*mail user agents* come **pine** o **kmail**) siano normalmente in grado di gestirli automaticamente, queste particolari utility consentono di manipolare tali allegati manualmente, da riga di comando, o in modalità batch per mezzo di uno script di shell.

crypt

Una volta questa era l'utility standard UNIX di cifratura di file. [4] Regolamenti governativi (USA N.d.T.), attuati per ragioni politiche, che proibiscono l'esportazione di software crittografico, hanno portato alla scomparsa di **crypt** da gran parte del mondo UNIX, nonché dalla maggioranza delle distribuzioni Linux. Per fortuna i programmatori hanno prodotto molte alternative decenti, tra le quali cruft (ftp://metalab.unc.edu/pub/Linux/utils/file/cruft-0.2.tar.gz) realizzata proprio dall'autore del libro (vedi Example A-5).

Miscellanea

mktemp

Crea un file temporaneo con nome "unico".

```
PREFISSO=nomefile
tempfile=‘mktemp $PREFISSO.XXXXXX‘
#                          ^^^^^^ Occorrono almeno 6 posti per
#+                                il suffisso del nome del file.
echo "nome del file temporaneo = $tempfile"
# nome del file temporaneo = nomefile.QA2ZpY
#                 o qualcosa del genere...
```

make

Utility per costruire e compilare pacchetti binari. Può anche essere usata per una qualsiasi serie di operazioni che devono essere eseguite a seguito di successive modifiche nei file sorgenti.

Il comando **make** verifica `Makefile`, che è un elenco di dipendenze ed operazioni che devono essere svolte.

install

Comando speciale per la copia di file. È simile a **cp**, ma in grado di impostare i permessi e gli attributi dei file copiati. Questo comando sembra fatto su misura per l'installazione di pacchetti software e come tale appare frequentemente nei `Makefile` (nella sezione *`make install`*). Potrebbe essere usato allo stesso modo in script d'installazione.

dos2unix

Questa utility, scritta da Benjamin Lin e collaboratori, converte i file di testo in formato DOS (righe che terminano con CR-LF) nel formato UNIX (righe che terminano con il solo LF), e viceversa.

ptx

Il comando **ptx [file-indicato]** produce un indice permutato (elenco a riferimento incrociato) del file. Questo, se necessario, può essere successivamente filtrato e ordinato in una pipe.

more
less

Comandi per visualizzare un file, o un flusso di testo, allo `stdout`, una schermata alla volta. Possono essere usati per filtrare l'output di uno script.

12.6. Comandi per comunicazioni

Alcuni dei comandi che seguono vengono utilizzati per la caccia agli spammer, così come per il trasferimento di dati e per le analisi della rete.

Informazioni e statistiche

host

Cerca informazioni su un host Internet per mezzo del nome o dell'indirizzo IP usando il DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

Ricerca informazioni su un indirizzo IP. Con l'opzione `-h`, **ipcalc** esegue una ricerca DNS inversa, per trovare il nome dell'host (server) a partire dall'indirizzo IP.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Esegue la "risoluzione del nome del server" di un host Internet per mezzo dell'indirizzo IP. Essenzialmente equivale a **ipcalc -h** o **dig -x**. Il comando può essere eseguito sia in modalità interattiva che non, vale a dire all'interno di uno script.

Il comando **nslookup** è stato immotivatamente "deprecato," ma viene ancora utilizzato.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
Server:         135.116.137.2
Address:        135.116.137.2#53

Non-authoritative answer:
Name:   kuhleersparnis.ch
```

dig

Simile a **nslookup**, esegue una "risoluzione del nome del server" Internet. Può essere eseguito sia in modalità interattiva che non, vale a dire da uno script.

Si confronti l'output di **dig -x** con **ipcalc -h** e **nslookup**.

```
bash$ dig -x 81.9.6.2
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;2.6.9.81.in-addr.arpa.         IN      PTR

;; AUTHORITY SECTION:
6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net. noc.eltel.net.
2002031705 900 600 86400 3600

;; Query time: 537 msec
;; SERVER: 135.116.137.2#53(135.116.137.2)
;; WHEN: Wed Jun 26 08:35:24 2002
;; MSG SIZE  rcvd: 91
```

traceroute

Traccia il percorso intrapreso dai pacchetti inviati ad un host remoto. Questo comando funziona in una LAN, WAN o su Internet. L'host remoto deve essere specificato per mezzo di un indirizzo IP. L'output può essere filtrato da grep o sed in una pipe.

```
bash$ traceroute 81.9.6.2
```

```
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
...
```

ping

Trasmette un pacchetto “ICMP ECHO_REQUEST” ad altre macchine, sia su rete locale che remota. È uno strumento diagnostico per verificare le connessioni di rete e dovrebbe essere usato con cautela.

Un **ping** che ha avuto successo restituisce exit status 0. Questo può essere verificato in uno script.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
Warning: time of day goes back, taking countermeasures.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709 usec
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286 usec

--- localhost.localdomain ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois

Esegue una ricerca DNS (Domain Name System). L’opzione `-h` consente di specificare quale server *whois* dev’essere interrogato. Vedi Example 4-6.

finger

Rintraccia informazioni sugli utenti di una rete. Opzionalmente, il comando può visualizzare i file `~/.plan`, `~/.project` e `~/.forward` di un utente, se presenti.

```
bash$ finger
Login  Name           Tty       Idle  Login Time   Office     Office Phone
bozo   Bozo Bozeman   tty1        8  Jun 25 16:59
bozo   Bozo Bozeman   ttyp0          Jun 25 16:59
bozo   Bozo Bozeman   ttyp1          Jun 25 17:07
```

```
bash$ finger bozo
Login: bozo                             Name: Bozo Bozeman
Directory: /home/bozo                   Shell: /bin/bash
On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
On since Fri Aug 31 20:13 (MST) on pts/1
```

```
On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
No mail.
No Plan.
```

Tralasciando considerazioni sulla sicurezza, molte reti disabilitano **finger** ed il demone ad esso associato. [5]

vrfy

Verifica un indirizzo e-mail Internet.

Accesso ad Host Remoto

sx
rx

La serie di comandi **sx** e **rx** serve a trasferire file a e da un host remoto utilizzando il protocollo *xmodem*. Generalmente sono compresi in un pacchetto comunicazioni, come **minicom**.

sz
rz

La serie di comandi **sz** e **rz** serve a trasferire file a e da un host remoto utilizzando il protocollo *zmodem*. *Zmodem* possiede alcuni vantaggi rispetto a *xmodem*, come una maggiore velocità di trasmissione e di ripresa di trasferimenti interrotti. Come **sx** e **rx**, generalmente sono compresi in un pacchetto comunicazioni.

ftp

Utility e protocollo per caricare/scaricare file su o da un host remoto. Una sessione ftp può essere automatizzata in uno script (vedi Example 17-6, Example A-5 ed Example A-14).

uucp

Copia da UNIX a UNIX. È un pacchetto per comunicazioni per il trasferimento di file tra server UNIX. Uno script di shell rappresenta un modo efficace per gestire una sequenza di comandi **uucp**.

Con l'avvento di Internet e della e-mail, **uucp** sembra essere precipitato nel dimenticatoio, ma esiste ancora e rimane perfettamente funzionante nelle situazioni in cui una connessione Internet non è adatta o non è disponibile.

cu

Chiama (*C*all *U*p) un sistema remoto e si connette come semplice terminale. Questo comando fa parte del pacchetto **uucp**. È una specie di versione inferiore di telnet.

telnet

Utility e protocollo di connessione ad host remoto.

> **Caution**
>
> Il protocollo telnet contiene buchi inerenti alla sicurezza e, quindi, dovrebbe essere evitato.

wget

L'utility **wget** recupera o scarica in modo *non-interattivo* file dal Web o da un sito ftp. Funziona bene in uno script.

```
wget -p http://www.xyz23.com/file01.html
wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -o $SAVEFILE
```

lynx

Brower per il Web ed i file. **lynx** può essere utilizzato all'interno di uno script (con l'opzione `-dump`) per recuperare un file dal Web o da un sito ftp in modalità non-interattiva.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

rlogin

`Remote login`, inizia una sessione su un host remoto. Dal momento che questo comando ha dei problemi inerenti alla sicurezza, al suo posto è meglio usare ssh.

rsh

`Remote shell`, esegue comandi su un host remoto. Anch'esso ha problemi di sicurezza. Si utilizzi, quindi, **ssh**.

rcp

`Remote copy`, copia file tra due differenti macchine collegate in rete. L'uso di **rcp** ed utility simili, aventi problemi di sicurezza, in uno script di shell, potrebbe non essere consigliabile. Si consideri, invece, l'utilizzo di **ssh** o di uno script **expect**.

ssh

`Secure shell`, si connette ad un host remoto e vi esegue dei comandi. Questo sostituto di sicurezza di **telnet**, **rlogin**, **rcp** e **rsh** utilizza l'autenticazione e la cifratura. Per i dettagli, si veda la sua pagina di manuale.

Rete Locale

write

È l'utility per la comunicazione terminale-terminale. Consente di inviare righe di testo dal vostro terminale (console o xterm) a quello di un altro utente. Si può usare, naturalmente, il comando mesg per disabilitare l'accesso di write in scrittura su di un terminale.

Poiché **write** è interattivo, normalmente non viene impiegato in uno script.

Mail

mail

Invia o legge messaggi e-mail.

Questo client per il recupero della posta da riga di comando funziona altrettanto bene come comando inserito in uno script.

Example 12-32. Uno script che si auto-invia

```
#!/bin/sh
# self-mailer.sh: Script che si auto-invia

adr=${1:-‘whoami‘}    # Imposta l’utente corrente come predefinito, se
      #+ non altrimenti specificato.
#  Digitando ’self-mailer.sh wiseguy@superdupergenius.com’
#+ questo script viene inviato a quel destinatario.
#  Il solo ’self-mailer.sh’ (senza argomento) invia lo script alla
#+ persona che l’ha invocato, per esempio, bozo@localhost.localdomain
#
#  Per i dettagli sul costrutto  ${parametro:-default}, vedi la sezione
#+ "Sostituzione di Parametro" del capitolo "Variabili Riviste".

# ============================================================================
  cat $0 | mail -s " Lo script \"‘basename $0‘\" si è auto-inviato." "$adr"
# ============================================================================

# --------------------------------------------------
#  Saluti dallo script che si auto-invia.
#  Una persona maliziosa ha eseguito questo script,
#+ che ne ha provocato l’invio a te. Apparentemente,
#+ certa gente non ha niente di meglio da fare
#+ con il proprio tempo.
# --------------------------------------------------

echo "Il ‘date‘, lo script \"‘basename $0‘\" è stato inviato a "$adr"."

exit 0
```

mailto

Simile al comando **mail**, **mailto** invia i messaggi e-mail da riga di comando o da uno script. Tuttavia, **mailto** consente anche l’invio di messaggi MIME (multimedia).

vacation

Questa utility risponde in automatico alle e-mail indirizzate ad un destinatario che si trova in vacanza o temporaneamente indisponibile. Funziona su una rete, in abbinamento con **sendmail**, e non è utilizzabile per un account di posta POP in dial-up.

12.7. Comandi di controllo terminale

Comandi riguardanti la console o il terminale

tput

Inizializza un terminale e/o ne recupera le informazioni dal database `terminfo`. Diverse opzioni consentono particolari operazioni sul terminale. **tput clear** è l'equivalente di **clear**, vedi oltre. **tput reset** è l'equivalente di **reset**, vedi oltre. **tput sgr0** annulla le impostazioni di un terminale, ma senza pulire lo schermo.

```
bash$ tput longname
xterm terminal emulator (XFree86 4.0 Window System)
```

L'esecuzione di **tput cup X Y** sposta il cursore alle coordinate (X,Y) nel terminale corrente. Normalmente dovrebbe essere preceduto dal comando **clear** per pulire lo schermo.

Si noti che stty offre una serie di comandi più potenti per il controllo di un terminale.

infocmp

Questo comando visualizza informazioni dettagliate sul terminale corrente. Utilizza, allo scopo, il database *terminfo*.

```
bash$ infocmp
#       Reconstructed via infocmp from file:
/usr/share/terminfo/r/rxvt
rxvt|rxvt terminal emulator (X Window System),
        am, bce, eo, km, mir, msgr, xenl, xon,
        colors#8, cols#80, it#8, lines#24, pairs#64,
        acsc=“aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, blink=\E[5m, bold=\E[1m,
        civis=\E[?25l,
        clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
        ...
```

reset

Annulla i parametri del terminale e pulisce lo schermo. Come nel caso di **clear**, il cursore ed il prompt vengono posizionati nell'angolo in alto a sinistra dello schermo.

clear

Il comando **clear** cancella semplicemente lo schermo di una console o di un xterm. Il prompt e il cursore riappaiono nell'angolo superiore sinistro dello schermo o della finestra xterm. Questo comando può essere usato sia da riga di comando che in uno script. Vedi Example 10-25.

script

Questa utility registra (salva in un file) tutte le digitazioni da riga di comando eseguite dall'utente su una console o in una finestra xterm. In pratica crea una registrazione della sessione.

12.8. Comandi per operazioni matematiche

"Calcoli matematici"

factor

Scompone un intero in fattori primi.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc

Bash non ha la possibilità di gestire i calcoli in virgola mobile, quindi non dispone di operatori per alcune importanti funzioni matematiche. Fortunatamente viene in soccorso **bc**.

Non semplicemente una versatile utility per il calcolo in precisione arbitraria, **bc** offre molte delle potenzialità di un linguaggio di programmazione.

bc possiede una sintassi vagamente somigliante al C.

Dal momento che si tratta di una utility che si comporta abbastanza bene su UNIX, e quindi può essere utilizzata in una pipe, **bc** risulta molto utile negli script.

Ecco un semplice modello di riferimento per l'uso di **bc** per calcolare una variabile di uno script. Viene impiegata la sostituzione di comando.

```
variabile=$(echo "OPZIONI; OPERAZIONI" | bc)
```

Example 12-33. Rata mensile di un mutuo

```
#!/bin/bash
# monthlypmt.sh: Calcola la rata mensile di un mutuo (prestito).


#  Questa è una modifica del codice del pacchetto "mcalc" (mortgage calculator),
#+ di Jeff Schmidt e Mendel Cooper (vostro devotissimo, autore di
#+ questo documento).
#  http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz  [15k]

echo
echo "Dato il capitale, il tasso d'interesse e la durata del mutuo,"
```

```
echo "calcola la rata di rimborso mensile."

denominatore=1.0

echo
echo -n "Inserisci il capitale (senza i punti di separazione)"
read capitale
echo -n "Inserisci il tasso d'interesse (percentuale)" # Se 12% inserisci "12",
                                                      #+ non ".12".
read t_interesse
echo -n "Inserisci la durata (mesi)"
read durata


 t_interesse=$(echo "scale=9; $t_interesse/100.0" | bc) #  Lo converte
                                                        #+ in decimale.
                  # "scale" determina il numero delle cifre decimali.


 tasso_interesse=$(echo "scale=9; $t_interesse/12 + 1.0" | bc)


 numeratore=$(echo "scale=9; $capitale*$tasso_interesse^$durata" | bc)

 echo; echo "Siate pazienti. È necessario un po' di tempo."

 let "mesi = $durata - 1"
# ==============================================================
 for ((x=$mesi; x > 0; x--))
 do
    bot=$(echo "scale=9; $tasso_interesse^$x" | bc)
    denominatore=$(echo "scale=9; $inferiore+$bot" | bc)
    #  denominatore = $(($denominatore + $bot"))
 done
# --------------------------------------------------------------
#  Rick Boivie ha indicato un'implementazione più efficiente del
#+ precedente ciclo che riduce di 2/3 il tempo di calcolo.

# for ((x=1; x <= $mesi; x++))
# do
#   denominatore=$(echo "scale=9; $denominatore * $tasso_interesse + 1" | bc)
# done


#  Dopo di che se n'è uscito con un'alternativa ancor più efficiente, una che
#+ abbatte il tempo di esecuzione di circa il 95%!

# denominatore='{
#  echo "scale=9; denominatore=$denominatore; tasso_interesse=$tasso_interesse"
#  for ((x=1; x <= $mesi; x++))
#  do
#       echo 'denominatore = denominatore * tasso_interesse + 1'
#  done
```

```
#  echo 'denominatore'
#  } | bc'       #  Ha inserito il 'ciclo for' all'interno di una
                 #+ sostituzione di comando.

# ============================================================================

 # let "rata = $numeratore/$denominatore"
 rata=$(echo "scale=2; $numeratore/$denominatore" | bc)
 # Vengono usate due cifre decimali per i centesimi di Euro.

 echo
 echo "rata mensile = Euro $rata"
 echo


 exit 0

 # Esercizi:
 #   1) Filtrate l'input per consentire l'inserimento del capitale con i
 #      punti di separazione.
 #   2) Filtrate l'input per consentire l'inserimento del tasso
 #      d'interesse sia in forma percentuale che decimale.
 #   3) Se siete veramente ambiziosi, implementate lo script per visualizzare
 #      il piano d'ammortamento completo.
```

Example 12-34. Conversione di Base

```
:
##############################################################################
# Shellscript:  base.sh - visualizza un numero in basi differenti (Bourne Shell)
# Autore     :  Heiner Steven (heiner.steven@odn.de)
# Data       :  07-03-95
# Categoria  :  Desktop
# $Id        :  base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
##############################################################################
# Descrizione
#
# Changes
# 21-03-95 stv        fixed error occuring with 0xb as input (0.2)
##############################################################################

# ==> Utilizzato in questo documento con il permesso dell'autore dello script.
# ==> Commenti aggiunti dall'autore del libro.

NOARG=65
NP='basename "$0"'                                  # Nome Programma
VER='echo '$Revision: 1.2 $' | cut -d' ' -f2'    # ==> VER=1.2

Utilizzo () {
    echo "$NP - visualizza un numero in basi diverse, $VER (stv '95)
utilizzo: $NP [numero ...]
```

```
Se non viene fornito alcun numero, questi vengono letti dallo standard input.
Un numero può essere
    binario (base 2)               inizia con 0b (es. 0b1100)
    ottale (base 8)                inizia con 0  (es. 014)
    esadecimale (base 16)          inizia con 0x (es. 0xc)
    decimale                       negli altri casi (es. 12)" >&2
    exit $NOARG
}   # ==> Funzione per la visualizzazione del messaggio di utilizzo.

Msg () {
    for i   # ==> manca in [lista].
    do echo "$NP: $i" >&2
    done
}

Fatale () { Msg "$@"; exit 66; }

VisualizzaBasi () {
    # Determina la base del numero
    for i      # ==> manca in [lista] ...
    do         # ==> perciò opera sugli argomenti forniti da riga di comando.
      case "$i" in
            0b*)               ibase=2;;        # binario
            0x*|[a-f]*|[A-F]*) ibase=16;;       # esadecimale
            0*)                ibase=8;;        # ottale
            [1-9]*)            ibase=10;;       # decimale
            *)

                 Msg "$i numero non valido - ignorato"
                 continue;;
      esac
      #  Toglie il prefisso, converte le cifre esadecimali in caratteri
      #+ maiuscoli (è richiesto da bc)
      numero=‘echo "$i" | sed -e ’s:^0[bBxX]::’ | tr ’[a-f]’ ’[A-F]’‘
      # ==> Si usano i ":" come separatori per sed, al posto della "/".

      # Converte il numero in decimale
      dec=‘echo "ibase=$ibase; $numero" | bc‘
      # ==> ’bc’ è l’utility di calcolo.
      case "$dec" in
      [0-9]*)        ;;                      # numero ok
      *)                continue;;           # errore: ignora
      esac

      # Visualizza tutte le conversioni su un’unica riga.
      # ==> ’here document’ fornisce una lista di comandi a ’bc’.
      echo ‘bc <<!
           obase=16; "esa="; $dec
           obase=10; "dec="; $dec
           obase=8;  "ott="; $dec
           obase=2;  "bin="; $dec
!
      ‘ | sed -e ’s: :        :g’
```

```
    done
}

while [ $# -gt 0 ]
do
    case "$1" in
            --)        shift; break;;
            -h)        Utilizzo;;            # ==> Messaggio di aiuto.
            -*)        Utilizzo;;
             *)        break;;               # primo numero
    esac   # ==> Sarebbe utile un'ulteriore verifica d'errore per un input
           #+    non consentito.
    shift
done

if [ $# -gt 0 ]
then
    VisualizzaBasi "$@"
else                                         # legge dallo stdin
    while read riga
    do
        VisualizzaBasi $riga
    done
fi
```

Un metodo alternativo per invocare **bc** comprende l'uso di un here document inserito in un blocco di sostituzione di comando. Questo risulta particolarmente appropriato quando uno script ha la necessità di passare un elenco di opzioni e comandi a **bc**.

```
variabile=‘bc << STRINGA_LIMITE
opzioni
enunciati
operazioni
STRINGA_LIMITE
‘

...oppure...


variabile=$(bc << STRINGA_LIMITE
opzioni
enunciati
operazioni
STRINGA_LIMITE
)
```

Example 12-35. Invocare bc usando un “here document”

```
#!/bin/bash
# Invocare ’bc’ usando la sostituzione di comando
# in abbinamento con un ’here document’.


var1=‘bc << EOF
18.33 * 19.78
EOF
’
echo $var1       # 362.56


#  $( ... ) anche questa notazione va bene.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Restituisce il seno di 1.7 radianti.
# L’opzione "-l" richiama la libreria matematica di ’bc’.
echo $var3       # .991664810


# Ora proviamolo in una funzione...
ip=              # Dichiarazione di variabile globale.
ipotenusa ()     # Calcola l’ipotenusa di un triangolo rettangolo.
{
ip=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
#  Sfortunatamente, non si può avere un valore di ritorno in virgola mobile
#+ da una funzione Bash.
}
```

```
ipotenusa 3.68 7.31
echo "ipotenusa = $ip"     # 8.184039344


exit 0
```

Example 12-36. Calcolare il pi greco

```
#!/bin/bash
# cannon.sh: Approssimare il PI a cannonate.

# È un esempio molto semplice della simulazione "Monte Carlo", un modello
#+ matematico di un evento reale, utilizzando i numeri pseudocasuali per
#+ emulare la probabilità dell'urna.

#  Consideriamo un appezzamento di terreno perfettamente quadrato, di 10000
#+ unità di lato.
#  Questo terreno ha, al centro, un lago perfettamente circolare con un
#+ diametro di 10000 unità.
#  L'appezzamento è praticamente tutta acqua, tranne i quattro angoli
#+ (Immaginatelo come un quadrato con inscritto un cerchio).
#
#  Spariamo delle palle con un vecchio cannone sul terreno quadrato. Tutti i
#+ proiettili cadranno in qualche parte dell'appezzamento, o nel lago o negli
#+ angoli emersi.
#  Poiché il lago occupa la maggior parte dell'area del terreno, la maggior
#+ parte dei proiettili CADRA' nell'acqua.
#  Solo pochi COLPIRANNO il terreno ai quattro angoli dell'appezzamento.
#
#  Se le cannonate sparate saranno sufficientemente casuali, senza aver
#+ mirato, allora il rapporto tra le palle CADUTE IN ACQUA ed il totale degli
#+ spari approssimerà il valore di PI/4.
#
#  La spiegazione sta nel fatto che il cannone spara solo al quadrante superiore
#+ destro del quadrato, vale a dire, il 1° Quadrante del piano di assi
#+ cartesiani. (La precedente spiegazione era una semplificazione.)
#
#  Teoricamente, più alto è il numero delle cannonate, maggiore
#+ sarà l'approssimazione.
#  Tuttavia, uno script di shell, in confronto ad un linguaggio compilato che
#+ dispone delle funzione matematiche in virgola mobile, richiede un po' di
#+ compromessi.
#  Sfortunatamente, questo fatto tende a diminuire la precisione della
#+ simulazione.


DIMENSIONE=10000 #  Lunghezza dei lati dell'appezzamento di terreno.
                 #  Imposta anche il valore massimo degli interi
 #+ casuali generati.

MAXSPARI=1000    #  Numero delle cannonate.
                 #  Sarebbe stato meglio 10000 o più, ma avrebbe
```

```
 #+ richiesto troppo tempo.
                #
PMULTIPL=4.0     # Fattore di scala per approssimare PI.

genera_casuale ()
{
SEME=$(head -1 /dev/urandom | od -N 1 | awk ’{ print $2 }’)
RANDOM=$SEME                           #  Dallo script di esempio
                                       #+ "seeding-random.sh".
let "rnum = $RANDOM % $DIMENSIONE"     #  Intervallo inferiore a 10000.
echo $rnum
}

distanza=        # Dichiarazione di variabile globale.
ipotenusa ()     # Calcola l’ipotenusa di un triangolo rettangolo.
{                # Dall’esempio "alt-bc.sh".
distanza=$(bc -l <<EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
#  Impostando "scale" a zero il risultato viene troncato (vengono eliminati i
#+ decimali), un compromesso necessario in questo script.
#  Purtroppo. questo diminuisce la precisione della simulazione.
}


# main() {

# Inizializzazione variabili.
spari=0
splash=0
terra=0
Pi=0

while [ "$spari" -lt  "$MAXSPARI" ]       #  Ciclo principale.
do

  xCoord=$(genera_casuale)                #  Determina le
     #+ coordinate casuali X e Y.
  yCoord=$(genera_casuale)
  ipotenusa $xCoord $yCoord               #  Ipotenusa del triangolo
     #+ rettangolo = distanza.
  ((spari++))

  printf "#%4d   " $spari
  printf "Xc = %4d  " $xCoord
  printf "Yc = %4d  " $yCoord
  printf "Distanza = %5d  " $distanza     #  Distanza dal centro del lago,
                                          #+ origine degli assi,
  #+ coordinate 0,0.

  if [ "$distanza" -le "$DIMENSIONE" ]
```

```
  then
      echo -n "SPLASH!  "
      ((splash++))
  else
      echo -n "TERRENO! "
      ((terra++))
  fi

  Pi=$(echo "scale=9; $PMULTIPL*$splash/$spari" | bc)
  # Moltiplica il rapporto per 4.0.
  echo -n "PI ~ $Pi"
  echo

done

echo
echo "Dopo $spari cannonate, $Pi sembra approssimare PI."
#  Tende ad essere un po' più alto . . .
#  Probabilmente a causa degli arrotondamenti e dell'imperfetta casualità di
#+ $RANDOM.
echo

# }

exit 0

#  Qualcuno potrebbe ben chiedersi se uno script di shell sia appropriato per
#+ un'applicazione così complessa e ad alto impiego di risorse qual'è una
#+ simulazione.
#
#  Esistono almeno due giustificazioni.
#  1) Come prova concettuale: per dimostrare che può essere fatto.
#  2) Per prototipizzare e verificare gli algoritmi prima della
#+    riscrittura in un linguaggio compilato di alto livello.
```

dc

L'utility **dc** (**d**esk **c**alculator) è orientata allo stack e usa la RPN ("Reverse Polish Notation"). Come **bc**, possiede molta della potenza di un linguaggio di programmazione.

La maggior parte delle persone evita **dc** perché richiede un input RPN non intuitivo. Viene comunque utilizzata.

Example 12-37. Convertire un numero decimale in esadecimale

```
#!/bin/bash
# hexconvert.sh: Converte un numero decimale in esadecimale.

BASE=16     # Esadecimale.

if [ -z "$1" ]
then
  echo "Utilizzo: $0 numero"
```

```
  exit $E_ERR_ARG
  # È necessario un argomento da riga di comando.
fi
# Esercizio: aggiungete un'ulteriore verifica di validità dell'argomento.


esacvt ()
{
if [ -z "$1" ]
then
  echo 0
  return    #  "Restituisce" 0 se non è stato passato nessun argomento alla
            #+ funzione.
fi

echo ""$1" "$BASE" o p" | dc
#                 "o" imposta la radice (base numerica) dell'output.
#                   "p" visualizza la parte alta dello stack.
# Vedi 'man dc' per le altre opzioni.
return
}

esacvt "$1"

exit 0
```

Lo studio della pagina *info* di **dc** fornisce alcuni chiarimenti sulle sue difficoltà . Sembra esserci, comunque, un piccolo, selezionato gruppo di *maghi di dc* che si deliziano nel mettere in mostra la loro maestria nell'uso di questa potente, ma arcana, utility.

Example 12-38. Fattorizzazione

```
#!/bin/bash
# factr.sh: Fattorizza un numero

MIN=2       # Non funzionerà con con un numero inferiore a questo.
E_ERR_ARG=65
E_INFERIORE=66

if [ -z $1 ]
then
  echo "Utilizzo: $0 numero"
  exit $E_ERR_ARG
fi

if [ "$1" -lt "$MIN" ]
then
  echo "Il numero da fattorizzare deve essere $MIN o maggiore."
  exit $E_INFERIORE
fi

#  Esercizio: Aggiungete una verifica di tipo (per rifiutare un argomento
#+ diverso da un intero).
```

```
echo "Fattori primi di $1:"
# -------------------------------------------------------------------------
echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2"\
| dc
# -------------------------------------------------------------------------
# La precedente riga di codice è stata scritta da Michel Charpentier
# <charpov@cs.unh.edu>.
# Usata con il permesso dell'autore (grazie).

exit 0
```

awk

Un altro modo ancora per eseguire calcoli in virgola mobile in uno script, è l'uso delle funzioni matematiche built-in di awk in uno shell wrapper.

Example 12-39. Calcolo dell'ipotenusa di un triangolo

```
#!/bin/bash
# hypotenuse.sh: Calcola l'"ipotenusa" di un triangolo rettangolo.
#               ( radice quadrata della somma dei quadrati dei cateti)

ARG=2               # Lo script ha bisogno che gli vengano passati i cateti
                    #+ del triangolo.
E_ERR_ARG=65        # Numero di argomenti errato.

if [ $# -ne "$ARG" ] # Verifica il numero degli argomenti.
then
  echo "Utilizzo: 'basename $0' cateto_1 cateto_2"
  exit $E_ERR_ARG
fi


SCRIPTAWK=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#            comando/i / parametri passati ad awk


echo -n "Ipotenusa di $1 e $2 = "
echo $1 $2 | awk "$SCRIPTAWK"

exit 0
```

12.9. Comandi diversi

Comandi che non possono essere inseriti in nessuna specifica categoria

jot
seq

Queste utility generano una sequenza di interi con un incremento stabilito dall'utente.

Il normale carattere di separazione tra ciascun intero è il ritorno a capo, che può essere modificato con l'opzione `-s`

```
bash$ seq 5
 1
 2
 3
 4
 5



bash$ seq -s : 5
1:2:3:4:5
```

Sia **jot** che **seq** si rivelano utili in un ciclo for.

Example 12-40. Utilizzo di seq per generare gli argomenti di un ciclo

```
#!/bin/bash
# Uso di "seq"

echo

for a in ‘seq 80‘  # oppure  for a in $( seq 80 )
#  Uguale a   for a in 1 2 3 4 5 ... 80   (si risparmia molta digitazione!).
#+ Si potrebbe anche usare ’jot’ (se presente nel sistema).
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
#  Esempio dell’uso dell’output di un comando per generare la [lista] di un
#+ ciclo "for".

echo; echo


CONTO=80  # Sì, ’seq’ può avere un parametro.

for a in ‘seq $CONTO‘  # o   for a in $( seq $CONTO )
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
```

```
echo; echo

INIZIO=75
FINE=80

for a in ‘seq $INIZIO $FINE‘
#  Fornendo due argomenti "seq" inizia il conteggio partendo dal primo e
#+ continua fino a raggiungere il secondo.
do
  echo -n "$a "
done      # 75 76 77 78 79 80

echo; echo

INIZIO=45
INTERVALLO=5
FINE=80

for a in ‘seq $INIZIO $INTERVALLO $FINE‘
#  Fornendo tre argomenti "seq" inizia il conteggio partendo dal primo, usa il
#+ secondo come passo (incremento) e continua fino a raggiungere il terzo.
do
  echo -n "$a "
done      # 45 50 55 60 65 70 75 80

echo; echo

exit 0
```

getopt

Il comando **getopt** verifica le opzioni, precedute da un trattino, passate da riga di comando. Questo comando esterno corrisponde al builtin di Bash getopts, ma non è altrettanto versatile. Tuttavia, usato con l’opzione `-l`, **getopt** permette la gestione delle opzioni estese.

Example 12-41. Utilizzo di getopt per verificare le opzioni passate da riga di comando

```
#!/bin/bash

# Provate ad invocare lo script nei modi seguenti:
#   sh ex33a.sh -a
#   sh ex33a.sh -abc
#   sh ex33a.sh -a -b -c
#   sh ex33a.sh -d
#   sh ex33a.sh -dXYZ
#   sh ex33a.sh -d XYZ
#   sh ex33a.sh -abcd
#   sh ex33a.sh -abcdZ
#   sh ex33a.sh -z
#   sh ex33a.sh a
```

```
# Spiegate i risultati di ognuna delle precedenti esecuzioni.

E_ERR_OPZ=65

if [ "$#" -eq 0 ]
then   # Lo script richiede almeno un argomento da riga di comando.
  echo "Utilizzo $0 -[opzioni a,b,c]"
  exit $E_ERR_OPZ
fi

set -- ‘getopt "abcd:" "$@"‘
# Imposta i parametri posizionali agli argomenti passati da riga di comando.
# Cosa succede se si usa "$*" invece di "$@"?

while [ ! -z "$1" ]
do
  case "$1" in
   -a) echo "Opzione \"a\"";;
   -b) echo "Opzione \"b\"";;
   -c) echo "Opzione \"c\"";;
   -d) echo "Opzione \"d\" $2";;
    *) break;;
  esac

  shift
done

#  Solitamente in uno script è meglio usare il builtin ’getopts’,
#+ piuttosto che ’getopt’.
#  Vedi "ex33.sh".

exit 0
```

run-parts

Il comando **run-parts** [6] esegue tutti gli script presenti nella directory di riferimento, sequenzialmente ed in ordine alfabetico. Naturalmente gli script devono avere i permessi di esecuzione.

Il demone crond invoca **run-parts** per eseguire gli script presenti nelle directory `/etc/cron.*`

yes

Il comportamento predefinito del comando **yes** è quello di inviare allo `stdout` una stringa continua del carattere y seguito da un ritorno a capo. **Control-c** termina l’esecuzione. Può essere specificata una diversa stringa di output, come **`yes altra stringa`** che visualizzerà in continuazione `altra stringa` allo `stdout`. Ci si può chiedere lo scopo di tutto questo. Sia da riga di comando che in uno script, l’output di **yes** può essere rediretto, o collegato per mezzo di una pipe, ad un programma in attesa di un input dell’utente. In effetti, diventa una specie di versione povera di **expect**

`yes | fsck /dev/hda1` esegue **fsck** in modalità non-interattiva (attenzione!).

`yes | rm -r nomedir` ha lo stesso effetto di **`rm -rf nomedir`** (attenzione!).

> **Warning**
>
> Si faccia soprattutto attenzione quando si collega, con una pipe, **yes** ad un comando di sistema potenzialmente pericoloso come fsck o fdisk. Potrebbero esserci degli effetti collaterali imprevisti.

banner

Visualizza gli argomenti allo `stdout` in forma di un ampio banner verticale, utilizzando un carattere ASCII (di default '#'), che può essere rediretto alla stampante per un hardcopy.

printenv

Visualizza tutte le variabili d'ambiente di un particolare utente.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

I comandi **lp** ed **lpr** inviano uno o più file alla coda di stampa per l'hardcopy. [7] I nomi di questi comandi derivano da "line printer", stampanti di un'altra epoca.

`bash$ lp file1.txt` o `bash lp <file1.txt`

Risulta spesso utile collegare a **lp**, con una pipe, l'output impaginato con **pr**.

`bash$ pr -opzioni file1.txt | lp`

Pacchetti di formato, quali **groff** e *Ghostscript*, possono inviare direttamente i loro output a **lp**.

`bash$ groff -Tascii file.tr | lp`

`bash$ gs -opzioni | lp file.ps`

Comandi correlati sono **lpq**, per visualizzare la coda di stampa, e **lprm**, per cancellare i job dalla coda di stampa.

tee

[Qui UNIX prende a prestito un'idea dall'idraulica.]

È un operatore di redirezione, ma con una differenza. Come il raccordo a "ti" (T) dell'idraulico, consente di "deviare" *in un file* l'output di uno o più comandi di una pipe, senza alterarne il risultato. È utile per registrare in un file, o in un documento, il comportamento di un processo, per tenerne traccia a scopo di debugging.

```
                  tee
                 |------> al file
                 |
  ===============|============================================
  comando--->----|-operatore-->---> risultato del/dei comando/i
```

```
============================================================
```

```
cat elencofile* | sort | tee file.verifica | uniq > file.finale
```

(Il file `file.verifica` contiene i file ordinati e concatenati di “elencofile”, prima che le righe doppie vengano cancellate da uniq.)

mkfifo

Questo oscuro comando crea una *named pipe*, un *buffer first-in-first-out* temporaneo, per il trasferimento di dati tra processi. [8] Tipicamente, un processo scrive nel FIFO e un altro vi legge. Vedi Example A-16.

pathchk

Questo comando verifica la validità del nome di un file. Viene visualizzato un messaggio d’errore nel caso in cui il nome del file ecceda la lunghezza massima consentita (255 caratteri), oppure quando una o più delle directory del suo percorso non vengono trovate.

Purtroppo, **pathchk** non restituisce un codice d’errore riconoscibile e quindi è praticamente inutile in uno script. Si prendano in considerazione, al suo posto, gli operatori di verifica di file.

dd

Questo è l’alquanto oscuro e molto temuto comando di “duplicazione dati”. Sebbene in origine fosse una utility per lo scambio di dati contenuti su nastri magnetici tra minicomputer UNIX e mainframe IBM, questo comando viene tuttora utilizzato. Il comando **dd** copia semplicemente un file (o lo `stdin/stdout`), ma con delle conversioni. Le conversioni possibili sono ASCII/EBCDIC, [9] maiuscolo/minuscolo, scambio di copie di byte tra input e output, e saltare e/o troncare la parte iniziale o quella finale di un file di input. **`dd --help`** elenca le conversioni e tutte le altre opzioni disponibili per questa potente utility.

```
# Esercitarsi con ’dd’.

n=3
p=5
file_input=progetto.txt
file_output=log.txt

dd if=$file_input of=$file_output bs=1 skip=$((n-1)) \
count=$((p-n+1)) 2> /dev/null
# Estrae i caratteri da n a p dal file $file_input.




echo -n "Ciao mondo" | dd cbs=1 conv=unblock 2> /dev/null
# Visualizza "Ciao mondo" verticalmente.


# Grazie, S.C.
```

Per dimostrare quanto versatile sia **dd** , la usiamo per catturare i tasti premuti.

Example 12-42. Intercettare i tasti premuti

```
#!/bin/bash
# Intercetta i tasti premuti senza dover premere anche INVIO.


tastipremuti=4                          # Numero di tasti da catturare.


precedenti_impostazioni_tty=$(stty -g) #  Salva le precedenti
                                       #+ impostazioni del terminale.

echo "Premi $tastipremuti tasti."
stty -icanon -echo                     # Disabilita la modalità canonica.
                                       # Disabilita l'eco locale.
tasti=$(dd bs=1 count=$tastipremuti 2> /dev/null)
# 'dd' usa lo stdin, se non viene specificato "if".

stty "$precedenti_impostazioni_tty"    # Ripristina le precedenti impostazioni.

echo "Hai premuto i tasti \"$tasti\"."

# Grazie, S.C. per la dimostrazione.
exit 0
```

Il comando **dd** può eseguire un accesso casuale su un flusso di dati.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
#  L'opzione "conv=notrunc" significa che il file di output non verrà troncato.

# Grazie, S.C.
```

Il comando **dd** riesce a copiare dati grezzi e immagini di dischi su e dai dispositivi, come floppy e dispositivi a nastro (Example A-6). Un uso comune è quello per creare dischetti di boot.

```
dd if=immagine-kernel of=/dev/fd0H1440
```

In modo simile, **dd** può copiare l'intero contenuto di un floppy, persino di uno formattato su un SO "straniero" , sul disco fisso come file immagine.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

Altre applicazioni di **dd** comprendono l'inizializzazione di file di swap temporanei (Example 29-2) e di ramdisk (Example 29-3). Può anche eseguire una copia di basso livello di un'intera partizione di un disco fisso, sebbene ciò non sia particolarmente raccomandabile.

Ci sono persone (presumibilmente che non hanno niente di meglio da fare con il loro tempo) che pensano costantemente ad applicazioni interessanti di **dd**.

Example 12-43. Cancellare in modo sicuro un file

```
#!/bin/bash
# blotout.sh: Cancella ogni traccia del file.

#  Questo script sovrascrive il file di riferimento alternativamente con byte
#+ casuali e con zeri, prima della cancellazione finale.
#  Dopo di che, anche un esame diretto dei settori del disco non riuscirà a
#+ rivelare i dati originari del file.

PASSI=7          #  Numero di sovrascritture.
DIMBLOCCO=1      #  L'I/O con /dev/urandom richiede di specificare la dimensione
                 #+ del blocco, altrimenti si ottengono risultati strani.
E_ERR_ARG=70
E_FILE_NON_TROVATO=71
E_CAMBIO_IDEA=72

if [ -z "$1" ]   # Nessun nome di file specificato.
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG
fi

file=$1

if [ ! -e "$file" ]
then
  echo "Il file \"$file\" non è stato trovato."
  exit $E_FILE_NON_TROVATO
fi

echo; echo -n "Sei assolutamente sicuro di voler cancellare \"$file\" (s/n)? "
read risposta
case "$risposta" in
[nN]) echo "Hai cambiato idea, vero?"
      exit $E_CAMBIO_IDEA
      ;;
*)    echo "Cancellazione del file \"$file\".";;
esac


dim_file=$(ls -l "$file" | awk '{print $5}')  #  Il 5° campo è la dimensione
                                              #+ del file.

conta_passi=1

echo

while [ "$conta-passi" -le "$PASSI" ]
do
  echo "Passaggio nr.$conta_passi"
  sync            # Scarica i buffer.
  dd if=/dev/urandom of=$file bs=$DIMBLOCCO count=$dim_file
```

```
                  # Sovrascrive con byte casuali.
  sync            # Scarica ancora i buffer.
  dd if=/dev/zero of=$file bs=$DIMBLOCCO count=$dim_file
                  # Sovrascrive con zeri.
  sync            # Scarica ancora una volta i buffer.
  let "conta_passi += 1"
  echo
done


rm -f $file    # Infine, cancella il file.
sync           # Scarica i buffer un'ultima volta.

echo "Il file \"$file\" è stato cancellato."; echo


#  È un metodo abbastanza sicuro, sebbene lento ed inefficiente, di rendere un
#+ file completamente "irriconoscibile".
#  Il comando "shred", che fa parte del pacchetto GNU "fileutils", esegue lo
#+ stesso lavoro, ma in maniera molto più efficiente.

#  La cancellazione non può essere "annullata" né il file recuperato con i
#+ metodi consueti.
#  Tuttavia... questo semplice metodo probabilmente *non* resisterebbe ad
#+ un'analisi forense.


#  Il pacchetto per la cancellazione sicura di file "wipe" di Tom Vier esegue
#+ un lavoro molto più completo di quanto non faccia questo semplice script.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

#  Per un'analisi approfondita sull'argomento della cancellazione sicura dei
#+ file, vedi lo studio di Peter Gutmann,
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#           http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html


exit 0
```

od

Il filtro **od**, ovvero *octal dump*, converte l'input (o i file) in formato ottale (base-8) o in altre basi. È utile per visualizzare o elaborare file dati binari o file di dispositivi di sistema altrimenti illeggibili, come `/dev/urandom`, e come filtro per i dati binari. Vedi Example 9-27 e Example 12-11.

hexdump

Esegue la conversione in esadecimale, ottale, decimale o ASCII di un file binario. Questo comando è grosso modo equivalente ad **od**, visto prima, ma non altrettanto utile.

objdump

Visualizza un file oggetto o un binario eseguibile sia in formato esadecimale che come listato assembly (con l'opzione `-d`).

```
bash$ objdump -d /bin/ls
/bin/ls:     file format elf32-i386

 Disassembly of section .init:

 080490bc <.init>:
  80490bc:       55                      push   %ebp
  80490bd:       89 e5                   mov    %esp,%ebp
  . . .
```

mcookie

Questo comando genera un "magic cookie", un numero esadecimale pseudocasuale di 128-bit (32-caratteri), normalmente usato come "firma" di autenticazione dal server X. È disponibile anche per gli script come mezzo "sbrigativo" per ottenere un numero casuale.

```
random000=‘mcookie | sed -e ’2p’‘
# Usa ’sed’ per eliminare i caratteri estranei.
```

Naturalmente, uno script potrebbe utilizzare per lo stesso scopo md5.

```
# Genera una checksum md5 dello script stesso.
random001=‘md5sum $0 | awk ’{print $1}’‘
# Usa ’awk’ per eliminare il nome del file.
```

Il comando **mcookie** fornisce un altro metodo, ancora, per generare un nome di file "unico" .

Example 12-44. Generatore di nomi di file

```
#!/bin/bash
# tempfile-name.sh:  generatore di nomi di file temporanei

STR_BASE=‘mcookie‘   # magic cookie di 32-caratteri.
POS=11               # Posizione arbitraria nella stringa magic cookie.
LUN=5                # Ottiene $LUN caratteri consecutivi.

prefisso=temp        #  È, dopo tutto, un file "temporaneo".
                     #  Per una maggiore "unicità", generate il prefisso del
                     #+ nome del file usando lo stesso metodo del
                     #+ suffisso, di seguito.

suffisso=${STR_BASE:POS:LUN}
                     #  Estrae una stringa di 5-caratteri, iniziando dall’11a
                     #+ posizione.
```

```
nomefile_temp=$prefisso.$suffisso
                     # Crea il nome del file.

echo "Nome del file temporaneo = "$nomefile_temp""

# sh tempfile-name.sh
# Nome del file temporaneo = temp.e19ea

exit 0
```

units

Questa utility esegue la conversione tra differenti unità di misura. Sebbene normalmente venga invocata in modalità interattiva, **units** può essere utilizzata anche in uno script.

Example 12-45. Convertire i metri in miglia

```
#!/bin/bash
# unit-conversion.sh


converte_unità ()  # Vuole come argomenti le unità da convertire.
{
  cf=$(units "$1" "$2" | sed --silent -e ’1p’ | awk ’{print $2}’)
  # Toglie tutto tranne il reale fattore di conversione.
  echo "$cf"
}

Unità1=miglia
Unità2=metri
fatt_conv =‘converte_unità $Unità1 $Unità2‘
quantità=3.73

risultato=$(echo $quantità*$fatt_conv | bc)

echo "Ci sono $risultato $Unità2 in $quantità $Unità1."

#  Cosa succede se vengono passate alla funzione unità di misura
#+ incompatibili, come "acri" e "miglia"?

exit 0
```

m4

Un tesoro nascosto, **m4** è un potente filtro per l’elaborazione di macro, [10] virtualmente un linguaggio completo. Quantunque scritto originariamente come pre-processore per *RatFor*, **m4** è risultato essere utile come utility indipendente. Infatti, **m4** combina alcune delle funzionalità di eval, tr e awk con le sue notevoli capacità di espansione di macro.

Nel numero dell'aprile 2002 di Linux Journal (http://www.linuxjournal.com) vi è un bellissimo articolo su **m4** ed i suoi impieghi.

Example 12-46. Utilizzo di m4

```
#!/bin/bash
# m4.sh: Uso del processore di macro m4

# Stringhe
stringa=abcdA01
echo "len($stringa)" | m4                           # 7
echo "substr($stringa,4)" | m4                      # A01
echo "regexp($stringa,[0-1][0-1],\&Z)" | m4         # 01Z

# Calcoli aritmetici
echo "incr(22)" | m4                                # 23
echo "eval(99 / 3)" | m4                            # 33

exit 0
```

doexec

Il comando **doexec** abilita il passaggio di un elenco di argomenti, di lunghezza arbitraria, ad un *binario eseguibile*. In particolare, passando `argv[0]` (che corrisponde a $0 in uno script), permette che l'eseguibile possa essere invocato con nomi differenti e svolgere una serie di azioni diverse, in accordo col nome con cui l'eseguibile è stato posto in esecuzione. Quello che si ottiene è un metodo indiretto per passare delle opzioni ad un eseguibile.

Per esempio, la directory `/usr/local/bin` potrebbe contenere un binario di nome "aaa". Eseguendo **doexec /usr/local/bin/aaa list** verrebbero elencati tutti quei file della directory di lavoro corrente che iniziano con una "a", mentre, (lo stesso eseguibile) con **doexec /usr/local/bin/aaa delete** quei file verrebbero *cancellati*.

> **Note:** I diversi comportamenti dell'eseguibile devono essere definiti nel codice dell'eseguibile stesso, qualcosa di analogo al seguente script di shell:
>
> ```
> case ‘basename $0‘ in
> "nome1" ) fa_qualcosa;;
> "nome2" ) fa_qualcosaltro;;
> "nome3" ) fa_un_altra_cosa_ancora;;
> *       ) azione_predefinita;;
> esac
> ```

dialog

La famiglia di strumenti dialog fornisce un mezzo per richiamare, da uno script, box di "dialogo" interattivi. Le varianti più elaborate di **dialog** -- **gdialog**, **Xdialog** e **kdialog** -- in realtà invocano i widget X-Windows. Vedi Example 34-15.

Notes

1. Questi sono file i cui nomi incominciano con un punto (dot), come `~/.Xdefaults`, e che non vengono visualizzati con un semplice **ls**. Non possono neanche essere cancellati accidentalmente con un **rm -rf ***. I dotfile vengono solitamente usati come file di impostazione e configurazione nella directory home dell'utente.
2. Questo è vero solo per la versione GNU di **tr**, non per la versione generica che si trova spesso sui sistemi commerciali UNIX.
3. **tar czvf nome_archivio.tar.gz *** *include* i dotfile presenti nelle directory che si trovano *al di sotto* della directory di lavoro corrente. Questa è una "funzionalità" non documentata del **tar** GNU.
4. Cifratura di tipo simmetrico, usata per i file su un sistema singolo o su una rete locale, contrapposta a quella a "chiave pubblica", di cui **pgp** è il ben noto esempio.
5. Un *demone* è un processo in esecuzione in background non collegato ad una sessione di terminale. I demoni eseguono servizi specifici sia ad ore indicate che al verificarsi di particolari eventi.

   La parola "demone" in greco significa fantasma, e vi è certamente qualcosa di misterioso, quasi soprannaturale, nel modo in cui i demoni UNIX vagano silenziosamente dietro le quinte eseguendo i compiti loro assegnati.
6. In realtà si tratta dell'adattamento di uno script della distribuzione Debian GNU/Linux.
7. La *coda di stampa* è il gruppo di job "in attesa" di essere stampato.
8. Per un'eccellente disamina di quest'argomento vedi l'articolo di Andy Vaught, Introduction to Named Pipes (http://www2.linuxjournal.com/lj-issues/issue41/2156.html), nel numero del Settembre 1997 di Linux Journal (http://www.linuxjournal.com).
9. EBCDIC (pronunciato "ebb-sid-ic") è l'acronimo di Extended Binary Coded Decimal Interchange Code. È un formato dati IBM non più molto usato. Una bizzarra applicazione dell'opzione `conv=ebcdic` di **dd** è la codifica, rapida e facile ma non molto sicura, di un file di testo.

   ```
   cat $file | dd conv=swab,ebcdic > $file_cifrato
   #  Codifica (lo rende inintelligibile).
   #  Si potrebbe anche fare lo switch dei byte (swab), per rendere la cosa un po'
   #+ più oscura.

   cat $file_cifrato | dd conv=swab,ascii > $file_testo
   # Decodifica.
   ```

10. Una *macro* è una costante simbolica che si espande in un comando o in una serie di operazioni sui parametri.

Chapter 13. Comandi di sistema e d’amministrazione

Gli script di avvio (startup) e di arresto (shutdown) presenti in `/etc/rc.d` illustrano gli usi (e l’utilità) di molti dei comandi che seguono. Questi, di solito, vengono invocati dall’utente root ed utilizzati per la gestione del sistema e per le riparazioni d’emergenza del filesystem. Vanno usati con attenzione poiché alcuni di questi comandi, se utilizzati in modo maldestro, possono danneggiare il sistema stesso.

Utenti e gruppi

users

Visualizza tutti gli utenti presenti sul sistema. Equivale approssimativamente a **who -q**.

groups

Elenca l’utente corrente ed i gruppi a cui appartiene. Corrisponde alla variabile interna $GROUPS, ma, anziché indicare i gruppi con i numeri corrispondenti, li elenca con i loro nomi.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

chown
chgrp

Il comando **chown** modifica la proprietà di uno o più file. Questo comando rappresenta un metodo utile che *root* può usare per spostare la proprietà di un file da un utente all’altro. Un utente ordinario non può modificare la proprietà dei file, neanche dei propri. [1]

```
root# chown bozo *.txt
```

Il comando **chgrp** modifica il *gruppo* proprietario di uno o più file. Occorre essere il proprietario del/dei file e membro del gruppo di destinazione (o *root*) per poter effettuare questa operazione.

```
chgrp --recursive dunderheads *.data
#  Il gruppo "dunderheads" adesso è proprietario di tutti i file"*.data"
#+ presenti nella directory $PWD (questo è il significato di "recursive").
```

useradd
userdel

Il comando d'amministrazione **useradd** aggiunge l'account di un utente al sistema e, se specificato, crea la sua directory home. Il corrispondente comando **userdel** cancella un utente dal sistema [2] ed i file ad esso associati.

> **Note:** Il comando **adduser** è il sinonimo di **useradd** nonché, di solito, un link simbolico ad esso.

id

Il comando **id** elenca i reali ID utente e di gruppo dell'utente corrente. È il corrispettivo delle variabili interne $UID, $EUID e $GROUPS.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

Vedi anche Example 9-5.

who

Visualizza tutti gli utenti connessi al sistema.

```
bash$ who
bozo  tty1     Apr 27 17:45
bozo  pts/0    Apr 27 17:46
bozo  pts/1    Apr 27 17:47
bozo  pts/2    Apr 27 17:49
```

L'opzione `-m` fornisce informazioni solo sull'utente corrente. Passare a **who** due argomenti, come nel caso di **who am i** o **who The Man** equivale a **who -m**.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami è simile a **who -m**, ma elenca semplicemente il nome dell'utente.

```
bash$ whoami
bozo
```

w

Visualizza tutti gli utenti connessi ed i processi di loro appartenenza. È la versione estesa di **who**. L'output di **w** può essere collegato con una pipe a **grep** per la ricerca di un utente e/o processo specifico.

```
bash$ w | grep startx
bozo  tty1     -                4:22pm  6:41   4.47s  0.45s  startx
```

logname

Visualizza il nome di login dell'utente corrente (così come si trova in `/var/run/utmp`). Equivale, quasi, al precedente whoami.

```
bash$ logname
bozo

bash$ whoami
bozo
```

Tuttavia...

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
bozo
```

su

Esegue un programma o uno script come utente diverso. **su rjones** esegue una shell come utente *rjones*. Il semplice **su** fa riferimento, in modo predefinito, all'utente *root*. Vedi Example A-16.

sudo

Esegue un comando come root (o altro utente). Può essere utilizzato in uno script, consentendone così l'esecuzione ad un utente ordinario.

```
#!/bin/bash

# Alcuni comandi.
sudo cp /root/secretfile /home/bozo/secret
# Ulteriori comandi.
```

Il file `/etc/sudoers` contiene i nomi degli utenti autorizzati ad invocare **sudo**.

passwd

Imposta o modifica la password dell'utente.

passwd può essere utilizzato in uno script, ma questo *non dovrebbe* essere fatto.

```
#!/bin/bash
#  set-new-password.sh: Non è una buona idea.
#  Questo script deve essere eseguito da root,
#+ o meglio ancora, non essere eseguito affatto!

ROOT_UID=0            # Root ha $UID 0.
E_UTENTE_ERRATO=65    # Non root?

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "Solo root può eseguire questo script."; echo
  exit $E_UTENTE_ERRATO
else
  echo; echo "Root, dovresti saper far di meglio che eseguire questo script."
fi

nomeutente=bozo
NUOVAPASSWORD=violazione_sicurezza

echo "$NUOVAPASSWORD" | passwd --stdin "$nomeutente"
#  L'opzione '--stdin' di 'passwd' permette
#+ di ottenere la nuova password dallo stdin (o da una pipe).

echo; echo "La password dell'utente $nomeutente è cambiata!"

# L'uso del comando 'passwd' in uno script è pericoloso.

exit 0
```

ac

Visualizza la durata della connessione di un utente al sistema, letta da `/var/log/wtmp`. Questa è una delle utility di contabilità GNU.

```
bash$ ac
        total       68.08
```

last

Elenca gli *ultimi* utenti connessi, letti da `/var/log/wtmp`. Questo comando consente anche la visualizzazione di login remoti.

newgrp

Modifica l'ID di gruppo dell'utente senza doversi disconnettere. Consente l'accesso ai file di un nuovo gruppo. Poiché gli utenti possono appartenere contemporaneamente a più gruppi, questo comando viene poco utilizzato.

Terminali

tty

Visualizza il nome del terminale dell’utente corrente. È da notare che ciascuna differente finestra di xterm viene considerata come un diverso terminale.

```
bash$ tty
/dev/pts/1
```

stty

Mostra e/o modifica le impostazioni del terminale. Questo complesso comando, usato in uno script, riesce a controllare il comportamento del terminale e le modalità di visualizzazione degli output. Si veda la sua pagina info e la si studi attentamente.

Example 13-1. Abilitare un carattere di cancellazione

```
#!/bin/bash
# erase.sh: Uso di "stty" per impostare un carattere di cancellazione nella
#+          lettura dell’input.

echo -n "Come ti chiami? "
read nome                       #  Provate ad usare tasto di ritorno (backspace)
                                #+ per cancellare i caratteri digitati.
                                #  Problemi?.
echo "Ti chiami $nome."

stty erase ’#’                  #  Imposta il carattere "hash" (#) come
                                #+ carattere di cancellazione.
echo -n "Come ti chiami? "
read nome                       #  Usate # per cancellare l’ultimo carattere
                                #+ digitato.
echo "Ti chiami $nome."

# Attenzione: questa impostazione rimane anche dopo l’uscita dallo script.

exit 0
```

Example 13-2. Password segreta: Disabilitare la visualizzazione a terminale

```
#!/bin/bash

echo
echo -n "Introduci la password "
read passwd
echo "La password è $passwd"
echo -n "Se qualcuno stesse sbirciando da dietro le vostre spalle,"
echo "la password sarebbe compromessa."

echo && echo  # Due righe vuote con una "lista and".
```

```
stty -echo    # Disabilita la visualizzazione sullo schermo.

echo -n "Reimmetti la password "
read passwd
echo
echo "La password è $passwd"
echo

stty echo     # Ripristina la visualizzazione sullo schermo.

exit 0
```

Un uso creativo di **stty** è quello di rilevare i tasti premuti dall’utente (senza dover premere successivamente **INVIO**).

Example 13-3. Rilevamento dei tasti premuti

```
#!/bin/bash
# keypress.sh: Rileva i tasti premuti dall’utente ("tastiera bollente").

echo

precedenti_impostazioni_tty=$(stty -g)  # Salva le precedenti impostazioni.
stty -icanon
tasti=$(head -c1)             #  Oppure $(dd bs=1 count=1 2> /dev/null)
                              #+ su sistemi non-GNU

echo
echo "Hai premuto i tasti \""$tasti"\"."
echo

stty "$precedenti_impostazioni_tty"     # Ripristina le precedenti impostazioni.

# Grazie, Stephane Chazelas.

exit 0
```

Vedi anche Example 9-3.

terminali e modalità

Normalmente, un terminale lavora in modalità *canonica*. Questo significa che quando un utente preme un tasto il carattere corrispondente non viene inviato immediatamente al programma in esecuzione in quel momento sul terminale. Un buffer specifico per quel terminale registra tutti i tasti premuti. Solo quando l'utente preme il tasto **INVIO** i caratteri digitati, che sono stati salvati nel buffer, vengono inviati al programma in esecuzione. All'interno di ciascun terminale è anche presente un elementare editor di linea.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>;
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
...
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

Utilizzando la modalità canonica è possibile ridefinire i tasti speciali dell'editor di riga del terminale.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>ciao mondo<ENTER>
<ctl-D>
bash$ cat filexxx
ciao mondo
bash$ bash$ wc -c < filexxx
11
```

Il processo che controlla il terminale riceve solamente 11 caratteri (10 alfabetici, più un ritorno a capo), sebbene l'utente abbia premuto 26 tasti.

In modalità non-canonica ("raw" -grezza), la pressione di ciascun tasto (compresi gli abbinamenti speciali come **ctl-H**) determina l'invio immediato del corrispondente carattere al processo di controllo.

Il prompt di Bash disabilita sia `icanon` che echo, dal momento che sostituisce l'editor di riga del terminale con un suo editor più elaborato. Così, per esempio, se si digita **ctl-A** al prompt della shell, non viene visualizza **^A** sullo schermo, Bash invece riceve il carattere **\1**, lo interpreta e sposta il cursore all'inizio della riga.

Stephane Chazelas

tset

Mostra o inizializza le impostazioni del terminale. È una versione meno potente di **stty**.

```
bash$ tset -r
Terminal type is xterm-xfree86.
Kill is control-U (^U).
Interrupt is control-C (^C).
```

setserial

Imposta o visualizza i parametri di una porta seriale. Questo comando deve essere eseguito dall'utente root e si trova, di solito, in uno script di avvio del sistema.

```
# Dallo script /etc/pcmcia/serial:

IRQ=‘setserial /dev/$DEVICE | sed -e ’s/.*IRQ: //’‘
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty
agetty

Il processo di inizializzazione di un terminale utilizza **getty** o **agetty** per l'impostazione del login di un utente. Questi comandi non vengono usati negli script di shell. Il loro corrispondente per lo scripting è **stty**.

mesg

Abilita o disabilita l'accesso in scrittura al terminale dell'utente corrente. Disabilitare l'accesso impedisce ad un altro utente della rete di scrivere su quel terminale.

> **Tip:** Può risultare molto fastidioso veder comparire improvvisamente un messaggio d'ordinazione di una pizza nel bel mezzo di un file di testo su cui si sta lavorando. Su una rete multi-utente, potrebbe essere desiderabile disabilitare l'accesso in scrittura al terminale quando si ha bisogno di evitare qualsiasi interruzione.

wall

È l'acronimo di " write all", vale a dire, inviare un messaggio ad ogni terminale di ciascun utente collegato alla rete. Si tratta, innanzi tutto, di uno strumento dell'amministratore di sistema, utile, per esempio, quando occorre avvertire tutti gli utenti che la sessione dovrà essere arrestata a causa di un determinato problema (vedi Example 17-2).

```
bash$ wall Tra 5 minuti Il sistema verrà sospeso per manutenzione!
Broadcast message from ecobel (pts/1) Sun Jul  8 13:53:27 2001...

Tra 5 minuti il sistema verrà sospeso per manutenzione!
```

> **Note:** Se l'accesso in scrittura di un particolare terminale è stato disabilitato con **mesg**, allora **wall** non potrà inviare nessun messaggio a quel terminale.

dmesg

Elenca allo `stdout` tutti i messaggi della fase di boot del sistema. Utile per il "debugging" e per verificare quali driver di dispositivo sono installati e quali interrupt vengono utilizzati. L'output di **dmesg** può, naturalmente, essere verificato con grep, sed o awk dall'interno di uno script.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
hda: IBM-DLGA-23080, ATA DISK drive
hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

Informazioni e statistiche

uname

Visualizza allo `stdout` le specifiche di sistema (SO, versione del kernel, ecc). Invocato con l'opzione `-a`, fornisce le informazioni in forma dettagliata (vedi Example 12-4). L'opzione `-s` mostra solo il tipo di Sistema Operativo.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown

bash$ uname -s
Linux
```

arch

Mostra l'architettura del sistema. Equivale a **uname -m**. Vedi Example 10-26.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Fornisce informazioni sui precedenti comandi, così come sono registrati nel file `/var/account/pacct`. Come opzioni si possono specificare il nome del comando e dell'utente. È una delle utility di contabilità GNU.

lastlog

Elenca l'ora dell'ultimo login di tutti gli utenti del sistema. Fa riferimento al file `/var/log/lastlog`.

```
bash$ lastlog
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
bin                                     **Never logged in**
daemon                                  **Never logged in**
```

```
...
bozo          tty1                      Sat Dec  8 21:14:29 -0700 2001


bash$ lastlog | grep root
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
```

> **Caution**
>
> Il comando fallisce se l’utente che l’ha invocato non possiede i permessi di lettura sul file `/var/log/lastlog`.

lsof

Elenca i file aperti. Questo comando visualizza una tabella dettagliata di tutti i file aperti in quel momento e fornisce informazioni sui loro proprietari, sulle dimensioni, sui processi ad essi associati ed altro ancora. Naturalmente, **lsof** può essere collegato tramite una pipe a grep e/o awk per verificare ed analizzare il risultato.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE  NODE NAME
init         1    root  mem    REG        3,5   30748    30303 /sbin/init
init         1    root  mem    REG        3,5   73120     8069 /lib/ld-2.1.3.so
init         1    root  mem    REG        3,5  931668     8075 /lib/libc-2.1.3.so
cardmgr    213    root  mem    REG        3,5   36956    30357 /sbin/cardmgr
...
```

strace

Strumento diagnostico e di debugging per il tracciamento dei segnali e delle chiamate di sistema. Il modo più semplice per invocarlo è **strace COMANDO**.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
uname({sys="Linux", node="bozo.localdomain", ...}) = 0
brk(0)                                  = 0x804f5e4
...
```

È l’equivalente Linux di **truss**.

nmap

Analizzatore delle porte di rete. Questo comando analizza un server per localizzare le porte aperte ed i servizi ad esse associati. È un importante strumento per la sicurezza, per proteggere una rete contro tentativi di hacking.

```
#!/bin/bash

SERVER=$HOST                            # localhost.localdomain (127.0.0.1).
NUMERO_PORTA=25                         # porta SMTP.

nmap $SERVER | grep -w "$NUMERO_PORTA"  # Questa specifica porta è aperta?
#              grep -w verifica solamente la parola esatta,
#+             così, per esempio, non verrà verificata la porta 1025.

exit 0

# 25/tcp     open        smtp
```

free

Mostra, in forma tabellare, l'utilizzo della memoria e della cache. L'output di questo comando è adatto alle verifiche per mezzo di grep, awk o **Perl**. Il comando **procinfo** visualizza tutte quelle informazioni che non sono fornite da **free**, e molto altro.

```
bash$ free
               total       used       free     shared buffers     cached
   Mem:         30504      28624       1880      15820    1608      16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

Per visualizzare la memoria RAM inutilizzata:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

Ricava ed elenca informazioni e statistiche dallo pseudo-filesystem `/proc`. Fornisce un elenco molto ampio e dettagliato.

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

Elenca i dispositivi, vale a dire, l'hardware installato.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
 ------------------------------------------------
cascade            4     2
```

```
dma                             0080-008f
dma1                            0000-001f
dma2                            00c0-00df
fpu                             00f0-00ff
ide0                      14  01f0-01f7 03f6-03f6
...
```

du

Mostra, in modo ricorsivo, l'utilizzo del (disco) file. Se non diversamente specificato, fa riferimento alla directory di lavoro corrente.

```
bash$ du -ach
1.0k    ./wi.sh
1.0k    ./tst.sh
1.0k    ./random.file
6.0k    .
6.0k    total
```

df

Mostra l'utilizzo del filesystem in forma tabellare.

```
bash$ df
Filesystem           1k-blocks      Used Available  Use% Mounted on
/dev/hda5               273262     92607    166547  36% /
/dev/hda8               222525    123951     87085  59% /home
/dev/hda7              1408796   1075744    261488  80% /usr
```

stat

Fornisce ampie e dettagliate *stat*istiche su un dato file (anche su una directory o su un file di dispositivo) o una serie di file.

```
bash$ stat test.cru
  File: "test.cru"
  Size: 49970        Allocated Blocks: 100          Filetype: Regular File
  Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/ bozo)
Device:  3,8   Inode: 18185     Links: 1
Access: Sat Jun  2 16:40:24 2001
Modify: Sat Jun  2 16:40:24 2001
Change: Sat Jun  2 16:40:24 2001
```

Se il file di riferimento non esiste, **stat** restituisce un messaggio d'errore.

```
bash$ stat file_inesistente
```

```
file_inesistente: No such file or directory
```

vmstat

Visualizza statistiche riguardanti la memoria virtuale.

```
bash$ vmstat
   procs                      memory    swap          io system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
 0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

netstat

Mostra informazioni e statistiche sulla rete corrente, come tabelle di routing e connessioni attive. Questa utility accede alle informazioni presenti in `/proc/net` (Chapter 28). Vedi Example 28-2.

netstat -r equivale a route.

uptime

Mostra da quanto tempo il sistema è attivo, con le relative statistiche.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

hostname

Visualizza il nome host del sistema. Questo comando imposta il nome dell'host in uno script di avvio in `/etc/rc.d` (`/etc/rc.d/rc.sysinit` o simile). Equivale a **uname -n** e corrisponde alla variabile interna $HOSTNAME.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

hostid

Visualizza un identificatore numerico esadecimale a 32 bit dell'host della macchina.

```
bash$ hostid
7f0100
```

Note: Si presume che questo comando possa fornire un numero seriale "unico" per un particolare sistema. Certe procedure per la registrazione di prodotto utilizzano questo numero per identificare una specifica licenza d'uso. Sfortunatamente, **hostid** restituisce solo l'indirizzo di rete della macchina in forma esadecimale con la trasposizione di una coppia di byte.

L'indirizzo di rete di una tipica macchina Linux, non appartenente ad una rete, si trova in `/etc/hosts`.

```
bash$ cat /etc/hosts
127.0.0.1               localhost.localdomain localhost
```

Si dà il caso che, con la trasposizione dei byte di **`127.0.0.1`**, si ottiene **`0.127.1.0`**, che trasformato in esadecimale corrisponde a **`007f0100`**, l'esatto equivalente di quanto è stato restituito da **hostid**, come visto in precedenza. Solo che esistono alcuni milioni di altre macchine Linux con questo stesso *hostid*.

sar

L'esecuzione di **sar** (System Activity Report) fornisce un dettagliatissimo resoconto delle statistiche di sistema. Santa Cruz Operation (SCO) ha rilasciato **sar** sotto licenza Open Source nel giugno 1999.

Questo comando non fa parte delle distribuzioni di base di Linux, ma è contenuto nel pacchetto sysstat utilities (http://perso.wanadoo.fr/sebastien.godard/), scritto da Sebastien Godard (mailto:sebastien.godard@wanadoo.fr).

```
bash$ sar
Linux 2.4.9 (brooks.seringas.fr)       09/26/03

10:30:00          CPU     %user     %nice   %system   %iowait     %idle
10:40:00          all      2.21     10.90     65.48      0.00     21.41
10:50:00          all      3.36      0.00     72.36      0.00     24.28
11:00:00          all      1.12      0.00     80.77      0.00     18.11
Average:          all      2.23      3.63     72.87      0.00     21.27

14:32:30          LINUX RESTART

15:00:00          CPU     %user     %nice   %system   %iowait     %idle
15:10:00          all      8.59      2.40     17.47      0.00     71.54
15:20:00          all      4.07      1.00     11.95      0.00     82.98
15:30:00          all      0.79      2.94      7.56      0.00     88.71
Average:          all      6.33      1.70     14.71      0.00     77.26
```

readelf

Mostra informazioni e statistiche sul file *elf* specificato. Fa parte del pacchetto *binutils*.

```
bash$ readelf -h /bin/bash
ELF Header:
   Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
   Class:                             ELF32
   Data:                              2's complement, little endian
   Version:                           1 (current)
```

```
   OS/ABI:                            UNIX - System V
   ABI Version:                       0
   Type:                              EXEC (Executable file)
   . . .
```

size

Il comando **size [/percorso/del/binario]** fornisce le dimensioni dei segmenti di un binario eseguibile o di un file archivio. È usato soprattutto dai programmatori.

```
bash$ size /bin/bash
   text    data     bss     dec     hex filename
 495971   22496   17392  535859   82d33 /bin/bash
```

Log di sistema

logger

Accoda messaggi generati dall’utente ai log di sistema (`/var/log/messages`). Non è necessario essere root per invocare **logger**.

```
logger Riscontrata un’instabilità nella connessione di rete alle 23:10, 05/21.
# Ora eseguite ’tail /var/log/messages’.
```

Inserendo il comando **logger** in uno script è possibile scrivere informazioni di debugging in `/var/log/messages`.

```
logger -t $0 -i Logging alla riga "$LINENO".
# L’opzione "-t" specifica l’identificativo della registrazione di logger
# L’opzione "-i" registra l’ID di processo.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging alla riga 3.
```

logrotate

Questa utility gestisce i file di log di sistema, effettuandone la rotazione, la compressione, la cancellazione e/o l’invio, secondo le necessità. Di solito crond esegue **logrotate** a cadenza giornaliera.

Aggiungendo una voce appropriata in `/etc/logrotate.conf` è possibile gestire i file di log personali allo stesso modo di quelli di sistema.

Controllo dei job

ps

Statistiche di processo (*P*rocess *S*tatistics): elenca i processi attualmente in esecuzione per proprietario e PID (ID di processo). Viene solitamente invocato con le opzioni `ax` e può essere collegato tramite una pipe a grep o sed per la ricerca di un processo specifico (vedi Example 11-11 e Example 28-1).

```
bash$  ps ax | grep sendmail
295 ?    S   0:00 sendmail: accepting connections on port 25
```

pstree

Elenca i processi attualmente in esecuzione in forma di struttura ad "albero" . L'opzione `-p` mostra i PID e i nomi dei processi.

top

Visualizza, in aggiornamento continuo, i processi maggiormente intensivi in termini di cpu. L'opzione `-b` esegue la visualizzazione in modalità testo, di modo che l'output possa essere verificato o vi si possa accedere da uno script.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
 Swap:  157208K av,       0K used,  157208K free                   37244K cached

 PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
   1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
   2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
   ...
```

nice

Esegue un job sullo sfondo (background) con priorità modificata. Le priorità vanno da 19 (la più bassa) a -20 (la più alta). Solo *root* può impostare le priorità negative (quelle più alte). Comandi correlati sono: **renice**, **snice** e **skill**.

nohup

Mantiene un comando in esecuzione anche dopo la sconnessione dell'utente. Il comando viene eseguito come un processo in primo piano (foreground) a meno che non sia seguito da &. Se si usa **nohup** in uno script, si prenda in considerazione di accoppiarlo a wait per evitare di creare un processo orfano o zombie.

pidof

Identifica l’*ID di processo (pid)* di un job in esecuzione. Poiché i comandi di controllo dei job, come kill e **renice**, agiscono sul *PID* di un processo (non sul suo nome), è necessario identificare quel determinato *PID*. Il comando **pidof** è approssimativamente simile alla variabile interna $PPID.

```
bash$ pidof xclock
880
```

Example 13-4. pidof aiuta ad terminare un processo

```
#!/bin/bash
# kill-process.sh

NESSUNPROCESSO=2

processo=xxxyyyzzz  # Si usa un processo inesistente.
# Solo a scopo dimostrativo...
# ... con questo script non si vuole terminare nessun processo in esecuzione.
#
# Se però voleste, per esempio, usarlo per scollegarvi da Internet, allora
#     processo=pppd

t=‘pidof $processo‘ # Cerca il pid (id di processo) di $processo.
#  Il pid è necessario a ’kill’ (non si può usare ’kill’ con
#+ il nome del programma).

if [ -z "$t" ]      # Se il processo non è presente, ’pidof’ restituisce null.
then
  echo "Il processo $processo non è in esecuzione."
  echo "Non è stato terminato alcun processo."
  exit $NESSUNPROCESSO
fi

kill $t             # Potrebbe servire ’kill -9’ per un processo testardo.

#  Qui sarebbe necessaria una verifica, per vedere se il processo ha
#+ acconsentito ad essere terminato.
#  Forse un altro " t=‘pidof $processo‘ ".


# L’intero script potrebbe essere sostituito da
#    kill $(pidof -x nome_processo)
# ma non sarebbe stato istruttivo.

exit 0
```

fuser

Identifica i processi (tramite il PID) che hanno accesso ad un dato file, serie di file o directory. Può anche essere invocato con l'opzione `-k` che serve a terminare quei determinati processi. Questo ha interessanti implicazioni per la sicurezza del sistema, specialmente negli script che hanno come scopo quello di evitare, agli utenti non autorizzati, l'accesso ai servizi di sistema.

crond

Programma schedulatore d'amministrazione che esegue determinati compiti, quali pulire e cancellare i file di log di sistema ed aggiornare il database slocate. È la versione superutente di at (sebbene ogni utente possa avere il proprio file `crontab` che può essere modificato con il comando **crontab**). Viene posto in esecuzione come demone ed esegue quanto specificato in `/etc/crontab`

Controllo di processo e boot

init

Il comando **init** è il genitore di tutti i processi. Richiamato nella parte finale della fase di boot, **init** determina il runlevel del sistema com'è specificato nel file `/etc/inittab`. Viene invocato per mezzo del suo alias **telinit** e solo da root.

telinit

Link simbolico a **init**, rappresenta il mezzo per modificare il runlevel del sistema che, di solito, è necessario per ragioni di manutenzione dello stesso o per riparazioni d'emergenza del filesystem. Può essere invocato solo da root. Questo comando è potenzialmente pericoloso - bisogna essere certi di averlo ben compreso prima di usarlo!

runlevel

Mostra l'ultimo, e attuale, runlevel, cioè se il sistema è stato fermato (runlevel `0`), se si trova in modalità utente singolo (`1`), in modalità multi-utente (`2` o `3`), in X Windows (`5`) o di riavvio (`6`). Questo comando ha accesso al file `/var/run/utmp`.

halt
shutdown
reboot

Serie di comandi per arrestare il sistema, solitamente prima dello spegnimento della macchina.

Rete

ifconfig

Utility per la configurazione e l'adattamento dell'interfaccia di rete. Viene usato molto spesso in fase di boot per impostare le interfacce, o per disabilitarle in caso di riavvio.

```
# Frammenti di codice dal file /etc/rc.d/init.d/network

# ...

# Controlla se la rete è attiva.
[ ${NETWORKING} = "no" ] && exit 0
```

```
[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
  if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
    action "L’interfaccia $i non è attiva: " ./ifdown $i boot
  fi
#  L’opzione "-q" di "grep", che è una specifica GNU, significa
#+ "quiet", cioè, non produce output.
#  Quindi, redirigere l’output in /dev/null non è strettamente necessario.

# ...

echo "Attualmente sono attivi questi dispositivi:"
echo ‘/sbin/ifconfig | grep ^[a-z] | awk ’{print $1}’‘
#                            ^^^^^
#           si dovrebbe usare il quoting per prevenire il globbing.
#  Anche le forme seguenti vanno bene.
#    echo $(/sbin/ifconfig | awk ’/^[a-z]/ { print $1 })’
#    echo $(/sbin/ifconfig | sed -e ’s/ .*//’)
#  Grazie, S.C. per i commenti aggiuntivi.
```

Vedi anche Example 30-6.

route

Mostra informazioni, o permette modifiche, alla tabella di routing del kernel.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
pm3-67.bozosisp*                255.255.255.255 UH       40 0          0 ppp0
127.0.0.0       *               255.0.0.0       U        40 0          0 lo
default         pm3-67.bozosisp 0.0.0.0         UG       40 0          0 ppp0
```

chkconfig

Verifica la configurazione di rete. Il comando elenca e gestisce i servizi di rete presenti nella directory `/etc/rc?.d` avviati durante il boot.

Trattandosi dell’adattamento fatto da Red Hat Linux dell’originario comando IRIX, **chkconfig** potrebbe non essere presente nell’installazione di base di alcune distribuzioni Linux.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
...
```

tcpdump

“Sniffa” i pacchetti di rete. È uno strumento per analizzare e risolvere problemi di traffico sulla rete per mezzo del controllo delle intestazioni di pacchetto che verificano criteri specifici.

Analizza gli ip dei pacchetti in transito tra gli host *bozoville* e *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Naturalmente, l’output di **tcpdump** può essere verificato usando alcune delle già trattate utility per l’elaborazione di testo.

Filesystem

mount

Monta un filesystem, solitamente di un dispositivo esterno, come il floppy disk o il CDROM. Il file `/etc/fstab` fornisce un utile elenco dei filesystem, partizioni e dispositivi disponibili, con le relative opzioni, che possono essere montati automaticamente o manualmente. Il file `/etc/mtab` mostra le partizioni e i filesystem attualmente montati (compresi quelli virtuali, come `/proc`).

mount -a monta tutti i filesystem e le partizioni elencate in `/etc/fstab`, ad eccezione di quelli con l’opzione `noauto`. Al boot uno script di avvio, presente in `/etc/rc.d` (`rc.sysinit` o qualcosa di analogo), invoca questo comando per montare tutto quello che deve essere montato.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Monta il CDROM
mount /mnt/cdrom
# Scorciatoia, se /mnt/cdrom è elencato in /etc/fstab
```

Questo versatile comando può persino montare un comune file su un dispositivo a blocchi, ed il file si comporterà come se fosse un filesystem. **Mount** riesce a far questo associando il file ad un dispositivo di loopback. Una sua possibile applicazione può essere quella di montare ed esaminare un’immagine ISO9660 prima di masterizzarla su un CDR. [3]

Example 13-5. Verificare un’immagine CD

```
# Da root...

mkdir /mnt/cdtest  # Prepara un punto di mount, nel caso non esistesse.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Monta l’immagine.
#                  l’opzione "-o loop" equivale a "losetup /dev/loop0"
cd /mnt/cdtest     # Ora verifica l’immagine.
ls -alR            # Elenca i file della directory.
                   # Eccetera.
```

umount

Smonta un filesystem attualmente montato. Prima di rimuovere fisicamente un floppy disk o un CDROM precedentemente montato, il dispositivo deve essere **smontato**, altrimenti si potrebbe ottenere, come risultato, la corruzione del filesystem.

```
umount /mnt/cdrom
# Ora potete premere il tasto eject e rimuovere in tutta sicurezza il disco.
```

> **Note:** L'utility **automount**, se correttamente installata, può montare e smontare i floppy disk e i CDROM nel momento in cui vi si accede o in fase di rimozione. Questa potrebbe, comunque, causare problemi sui portatili con dispositivi floppy e CDROM intercambiabili.

sync

Forza la scrittura immediata di tutti i dati aggiornati dai buffer all'hard disk (sincronizza l'HD con i buffer). Sebbene non strettamente necessario, **sync** assicura l'amministratore di sistema, o l'utente, che i dati appena modificati sopravviveranno ad un'improvvisa mancanza di corrente. Una volta, un **`sync; sync`** (due volte, tanto per essere assolutamente sicuri) era un'utile misura precauzionale prima del riavvio del sistema.

A volte può essere desiderabile una pulizia immediata dei buffer, come nel caso della cancellazione di sicurezza di un file (vedi Example 12-43) o quando le luci di casa incominciano a tremolare.

losetup

Imposta e configura i dispositivi di loopback.

Example 13-6. Creare un filesystem in un file

```
DIMENSIONE=1000000  # 1 mega

head -c $DIMENSIONE < /dev/zero > file  #  Imposta il file alla
                                        #+ dimensione indicata.
losetup /dev/loop0 file                 #  Lo imposta come dispositivo
                                        #+ di loopback.
mke2fs /dev/loop0                       # Crea il filesystem.
mount -o loop /dev/loop0 /mnt           # Lo monta.

# Grazie, S.C.
```

mkswap

Crea una partizione o un file di scambio. L'area di scambio dovrà successivamente essere abilitata con **swapon**.

swapon
swapoff

Abilita/disabilita una partizione o un file di scambio. Questi comandi vengono solitamente eseguiti in fase di boot o di arresto del sistema.

mke2fs

Crea un filesystem Linux di tipo ext2. Questo comando deve essere invocato da root.

Example 13-7. Aggiungere un nuovo hard disk

```
#!/bin/bash

# Aggiunge un secondo hard disk al sistema.
# Configurazione software. Si assume che l’hardware sia già montato sul PC.
# Da un articolo dell’autore di questo libro.
# Pubblicato sul nr. 38 di "Linux Gazette", http://www.linuxgazette.com.

ROOT_UID=0     # Lo script deve essere eseguito da root.
E_NONROOT=67   # Errore d’uscita non-root.

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Devi essere root per eseguire questo script."
  exit $E_NONROOT
fi

#  Da usare con estrema attenzione!
#  Se qualcosa dovesse andare storto, potreste cancellare irrimediabilmente
#+ il filesystem corrente.


NUOVODISCO=/dev/hdb        # Si assume che sia libero /dev/hdb. Verificate!
MOUNTPOINT=/mnt/nuovodisco # Oppure scegliete un altro punto di montaggio.

fdisk $NUOVODISCO
mke2fs -cv $NUOVODISCO1    #  Verifica i blocchi difettosi visualizzando un
                           #+ output dettagliato.
#  Nota:    /dev/hdb1, *non* /dev/hdb!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT     # Rende il nuovo disco accessibile a tutti gli utenti.


# Ora, una verifica...
# mount -t ext2 /dev/hdb1 /mnt/nuovodisco
# Provate a creare una directory.
# Se l’operazione riesce, smontate la partizione e procedete.

# Passo finale:
# Aggiungete la riga seguente in /etc/fstab.
# /dev/hdb1  /mnt/nuovodisco  ext2  defaults  1 1

exit 0
```

Vedi anche Example 13-6 e Example 29-3.

tune2fs

Serve per la taratura di un filesystem di tipo ext2. Può essere usato per modificare i parametri del filesystem, come il numero massimo dei mount. Deve essere invocato da root.

> **Warning**
>
> Questo è un comando estremamente pericoloso. Si usa a proprio rischio, perché si potrebbe inavvertitamente distruggere il filesystem.

dumpe2fs

Fornisce (elenca allo `stdout`) informazioni dettagliatissime sul filesystem. Dev'essere invocato da root.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Mount count:              6
Maximum mount count:      20
```

hdparm

Elenca o modifica i parametri dell'hard disk. Questo comando va invocato da root e può risultare pericoloso se usato in modo maldestro.

fdisk

Crea o modifica la tabella delle partizioni di un dispositivo per la registrazione dei dati, di solito un hard disk. Dev'essere invocato da root.

> **Warning**
>
> Si utilizzi questo comando con estrema attenzione. Se qualcosa dovesse andare storto si potrebbe distruggere il filesystem.

fsck
e2fsck
debugfs

Serie di comandi per la verifica, riparazione e "debugging" del filesystem.

fsck: front end per la verifica di un filesystem UNIX (può invocare altre utility). Il filesystem preimpostato, generalmente, è di tipo ext2.

e2fsck: esegue la verifica di un filesystem di tipo ext2.

debugfs: per il "debugging" di un filesystem di tipo ext2. Uno degli usi di questo versatile, ma pericoloso, comando è quello di (cercare di) recuperare i file cancellati. Solo per utenti avanzati!

> **Caution**
>
> Tutti i precedenti comandi dovrebbero essere invocati da root e, se usati in modo scorretto, potrebbero danneggiare o distruggere il filesystem.

badblocks

Verifica i blocchi difettosi (difetti fisici) di un dispositivo di registrazione dati. Questo comando viene usato per formattare un nuovo hard disk installato o per verificare l'integrità di un dispositivo per il backup. [4] Ad esempio, **badblocks /dev/fd0** verifica il floppy disk.

Il comando **badblocks** può essere invocato o in modalità distruttiva (sovrascrittura di tutti i dati) o non distruttiva, in sola lettura. Se l'utente root possiede il dispositivo che deve essere verificato, com'è di solito il caso, allora è root che deve invocare questo comando.

mkbootdisk

Crea un dischetto di boot che può essere usato per avviare il sistema se, per esempio, il MBR (master boot record) si è corrotto. Il comando **mkbootdisk**, in realtà, è uno script Bash scritto da Erik Troan che si trova nella directory `/sbin`.

chroot

Modifica la directory ROOT (CHange ROOT). Normalmente i comandi relativi a `/`, la directory root predefinita, vengono forniti da $PATH. Questo comando cambia la directory root predefinita in un'altra (che diventa anche la directory di lavoro corrente). Questo è utile per motivi di sicurezza, ad esempio quando l'amministratore di sistema desidera limitare l'attività di certi utenti, come quelli che stanno usando telnet, ad una porzione sicura del filesystem (talvolta si fa riferimento a questa azione come "confinare un utente in una prigione, o gabbia, chroot"). Si noti che dopo un chroot l'originario percorso degli eseguibili di sistema non è più valido.

Il comando **`chroot /opt`** dovrebbe cambiare il riferimento da `/usr/bin` in `/opt/usr/bin`. Allo stesso modo, **`chroot /aaa/bbb /bin/ls`** dovrebbe redirigere le successive chiamate di **ls** a `/aaa/bbb` come directory base, al posto di `/` com'è normalmente il caso. La riga **alias XX 'chroot /aaa/bbb ls'** inserita nel file `~/.bashrc` di un utente, delimita la porzione di filesystem (`/aaa/bbb`) sulla quale quell'utente può eseguire il comando "XX".

Il comando **chroot** è anche utile durante l'esecuzione da un dischetto di boot d'emergenza (**chroot** a `/dev/fd0`), o come opzione di **lilo** in caso di ripristino dopo un crash del sistema. Altri usi comprendono l'installazione da un filesystem diverso (un'opzione rpm) o l'esecuzione di un filesystem in sola lettura da CDROM. Va invocato solo da root ed usato con attenzione.

> **Caution**
>
> Potrebbe rendersi necessario copiare alcuni file di sistema nella directory indicata a *chroot* perché, dopo, non ci si potrà più basare sull'usuale variabile `$PATH`.

lockfile

Questa utility fa parte del pacchetto **procmail** (www.procmail.org (http://www.procmail.org)). Serve a creare un *file lock*, un semaforo che controlla l’accesso ad un file, ad un dispositivo o ad una risorsa. Il file lock sta ad indicare che quel particolare file, dispositivo o risorsa è utilizzato da un determinato processo (“busy”) e questo consente un accesso limitato (o nessun accesso) ad altri processi.

I file lock vengono utilizzati, ad esempio, per proteggere le cartelle di posta di sistema da modifiche fatte simultaneamente da più utenti, per indicare che si è avuto accesso ad una porta modem o per mostrare che un’istanza di Netscape sta usando la sua cache. È possibile, per mezzo di script, accertarsi dell’esistenza di un file lock creato da un certo processo, per verificare se quel processo è ancora in esecuzione. Si noti che se uno script cerca di creare un file lock già esistente, lo script, probabilmente, si bloccherà.

Normalmente, le applicazioni creano e verificano i file lock nella directory `/var/lock`. Uno script può accertarsi della presenza di un file lock con qualcosa di simile a quello che segue.

```
nomeapplicazione=xyzip
# L’applicazione "xyzip" ha creato il file lock "/var/lock/xyzip.lock".

if [ -e "/var/lock/$nomeapplicazione.lock ]

then
  ...
```

mknod

Crea file di dispositivo a blocchi o a caratteri (potrebbe essere necessario per l’installazione di nuovo hardware sul sistema).

tmpwatch

Cancella automaticamente i file a cui non si è acceduto da un determinato periodo di tempo. È invocato, di solito, da crond per cancellare vecchi file di log.

MAKEDEV

Utility per la creazione di file di dispositivo. Deve essere eseguito da root e ci si deve trovare nella directory `/dev`.

```
root# ./MAKEDEV
```

È una specie di versione avanzata di **mknod**.

Backup

dump
restore

Il comando **dump** è un’elaborata utility per il backup del filesystem e viene generalmente usata su installazioni e reti di grandi dimensioni. [5] Legge le partizioni del disco e scrive un file di backup in formato binario. I file di cui si deve eseguire il backup possono essere salvati sui più vari dispositivi di registrazione, compresi dischi e dispositivi a nastro. Il comando **restore** ripristina i backup effettuati con **dump**.

fdformat

Esegue una formattazione a basso livello di un dischetto.

Risorse di sistema

ulimit

Imposta un *limite superiore* all’uso delle risorse di sistema. Viene solitamente invocato con l’opzione `-f`, che imposta la dimensione massima del file (**ulimit -f 1000** limita la dimensione massima dei file a 1 mega). L’opzione `-t` imposta il limite dei file core (**ulimit -c 0** elimina i file core). Di norma, il valore di **ulimit** dovrebbe essere impostato nel file `/etc/profile` e/o `~/.bash_profile` (vedi Chapter 27).

Important: Un uso giudizioso di **ulimit** può proteggere il sistema contro una temibile *bomba fork*.

```
#!/bin/bash
# Script a solo scopo illustrativo.
# L’esecuzione è a vostro rischio -- vi *bloccherà* il sistema.

while true  #  Ciclo infinito.
do
  $0 &      #  Lo script invoca se stesso . . .
            #+ genera il processo un numero infinito di volte . . .
            #+ finché il sistema non si blocca a seguito
            #+ dell’esaurimento di tutte le risorse.
done        #  Questo è il famigerato scenario dell’“apprendista stregone”.

exit 0      #  Non esce qui, perché questo script non terminerà mai.
```

La riga **ulimit -Hu XX** (dove *XX* è il limite del processo utente), inserita nel file `/etc/profile`, avrebbe fatto abortire lo script appena lo stesso avesse superato il suddetto limite.

umask

Crea la MASCHERA per i file dell’utente. Riduce gli attributi predefiniti dei file di un particolare utente. Tutti i file creati da quell’utente otterranno gli attributi specificati con **umask**. Il valore (ottale) passato ad **umask** definisce i permessi *disabilitati* del file. Per esempio, **umask 022** fa sì che i nuovi file avranno al massimo i permessi 755 (777 NAND 022). [6] Naturalmente l’utente potrà, successivamente, modificare gli attributi di file particolari con chmod. È pratica corrente impostare il valore di **umask** in `/etc/profile` e/o `~/.bash_profile` (vedi Chapter 27).

rdev

Fornisce informazioni o esegue modifiche sulla partizione di root, sullo spazio di scambio (swap) o sulle modalità video. Le sue funzionalità sono state, in genere, superate da **lilo**, ma **rdev** resta utile per impostare un ram disk. Questo comando, se usato male, è pericoloso.

Moduli

lsmod

Elenca i moduli del kernel installati.

```
bash$ lsmod
Module                  Size  Used by
autofs                  9456   2 (autoclean)
opl3                   11376   0
serial_cs               5456   0 (unused)
sb                     34752   0
uart401                 6384   0 [sb]
sound                  58368   0 [opl3 sb uart401]
soundlow                 464   0 [sound]
soundcore               2800   6 [sb sound]
ds                      6448   2 [serial_cs]
i82365                 22928   2
pcmcia_core            45984   0 [serial_cs ds i82365]
```

Note: Le stesse informazioni si ottengono con **cat /proc/modules**.

insmod

Forza l’installazione di un modulo del kernel (quando è possibile è meglio usare **modprobe**). Deve essere invocato da root.

rmmod

Forza la disinstallazione di un modulo del kernel. Deve essere invocato da root.

modprobe

Carica i moduli ed è, solitamente, invocato automaticamente in uno script di avvio. Deve essere invocato da root.

depmod

Crea il file delle dipendenze dei moduli, di solito invocato da uno script di avvio.

Miscellanea

env

Esegue un programma, o uno script, impostando o modificando determinate variabili d’ambiente (senza dover modificare l’intero ambiente del sistema). `[nomevariabile=xxx]` consente di modificare la variabile d’ambiente `nomevariabile` per la durata dello script. Se non viene specificata nessuna opzione, questo comando elenca le impostazioni di tutte le variabili d’ambiente.

Note: In Bash e in altre shell derivate dalla Bourne, è possibile impostare le variabili nell'ambiente di un singolo comando.

```
var1=valore1 var2=valore2 comandoXXX
# $var1 e $var2 vengono impostate solo nell'ambiente di 'comandoXXX'.
```

Tip: È possibile usare **env** nella prima riga di uno script (la c.d.riga "sha-bang") quando non si conosce il percorso della shell o dell'interprete.

```
#! /usr/bin/env perl

print "Questo script Perl verrà eseguito,\n";
print "anche quando non sai dove si trova l'interprete Perl.\n";

# Ottimo per la portabilità degli script su altre piattaforme,
# dove i binari Perl potrebbero non essere dove ci aspettiamo.
# Grazie, S.C.
```

ldd

Mostra le dipendenze delle librerie condivise di un file eseguibile.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

Esegue un comando ripetutamente, ad intervalli di tempo specificati.

Gli intervalli preimpostati sono di due secondi, ma questo valore può essere modificato mediante l'opzione `-n`.

```
watch -n 5 tail /var/log/messages
#  Visualizza la parte finale del file di log di sistema /var/log/messages
#+ ogni cinque secondi.
```

strip

Rimuove i riferimenti simbolici per il "debugging" da un binario eseguibile. Questo diminuisce la sua dimensione, ma rende il "debugging" impossibile.

Questo comando si trova spesso nei Makefile, ma raramente in uno script di shell.

nm

Elenca i riferimenti simbolici, se non tolti con strip, presenti in un binario compilato.

rdist

Client per la distribuzione remota di file: sincronizza, clona o esegue il backup di un filesystem su un server remoto.

Utilizzando le conoscenze fin qui conseguite sui comandi d’amministrazione, ora si passa all’esame di uno script di sistema. Uno dei più brevi e più semplici da capire è **killall** che è utilizzato per sospendere i processi nella fase di arresto del sistema.

Example 13-8. killall, da `/etc/rc.d/init.d`

```
#!/bin/sh

# --> I commenti aggiunti dall’autore del libro sono indicati con "# -->".

# --> Questo fa parte del pacchetto di script ’rc’
# --> di Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>

# --> Sembra che questo particolare script sia specifico di Red Hat
# --> (potrebbe non essere presente in altre distribuzioni).

# Bring down all unneeded services that are still running (there shouldn’t
# be any, so this is just a sanity check)

for i in /var/lock/subsys/*; do
        # --> Ciclo standard for/in, ma poiché "do" è posto sulla stessa riga,
        # --> è necessario aggiungere il ";".
        # Check if the script is there.
        [ ! -f $i ] && continue
        # --> Ecco un uso intelligente di una "lista and", equivale a:
        # --> if [ ! -f "$i" ]; then continue

        # Get the subsystem name.
        subsys=${i#/var/lock/subsys/}
        # --> Imposta la variabile, in questo caso, al nome del file.
        # --> È l’equivalente esatto di subsys=‘basename $i‘.

        # -->  Viene ricavato dal nome del file lock (se esiste un file lock
        # -->+ che rappresenti la prova che il processo è in esecuzione).
        # -->  Vedi la precedente voce "lockfile".


        # Bring the subsystem down.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
            /etc/rc.d/init.d/$subsys.init stop
        else
            /etc/rc.d/init.d/$subsys stop
        # --> Sospende i job ed i demoni in esecuzione
        # --> usando il builtin di shell ’stop’.
fi
```

```
done
```

Non è poi così difficile. Tranne che per una piccola ed insolita impostazione di variabile, non vi è niente che già non si conosca.

Esercizio 1. Si analizzi lo script **halt** in `/etc/rc.d/init.d`. È leggermente più lungo di **killall**, ma concettualmente simile. Si faccia una copia dello script nella directory personale e con esso si eseguano delle prove (*non* va eseguito da root). Si effettui un'esecuzione simulata con le opzioni `-vn` (**`sh -vn nomescript`**). Si aggiungano commenti dettagliati. Si sostituiscano i comandi "action" con degli "echo".

Esercizio 2. Si dia un'occhiata ad alcuni degli script più complessi presenti in `/etc/rc.d/init.d`. Si veda se si riesce a comprendere parti di questi script. Per l'analisi, si segua la procedura spiegata nell'esercizio precedente. Per alcuni ulteriori approfondimenti si potrebbe anche esaminare il file `sysvinitfiles` in `/usr/share/doc/initscripts-?.??` che fa parte della documentazione "initscripts".

Notes

1. Questo è il caso su una macchina Linux o un sistema UNIX su cui è attiva la gestione delle quote del/dei disco/hi.
2. Il comando **userdel** non funziona se l'utente che deve essere cancellato è ancora connesso.
3. Per maggiori dettagli sulla registrazione dei CDROM, vedi l'articolo di Alex Withers, Creating CDs (http://www2.linuxjournal.com/lj-issues/issue66/3335.html), nel numero dell'Ottobre 1999 di Linux Journal (http://www.linuxjournal.com).
4. Anche il comando mke2fs con l'opzione `-c` esegue la verifica dei blocchi difettosi.
5. Gli operatori su sistemi Linux in modalità utente singolo, generalmente preferiscono qualcosa si più semplice per i backup, come **tar**.
6. NAND è l'operatore logico "not-and". La sua azione è paragonabile ad una sottrazione.

Chapter 14. Sostituzione di comando

La **sostituzione di comando** riassegna il risultato di un comando [1], o anche di più comandi. Letteralmente: connette l'output di un comando ad un altro contesto.

La forma classica di sostituzione di comando utilizza gli apici singoli inversi (‘...‘). I comandi all'interno degli apici inversi (apostrofi inversi) generano una riga di testo formata dai risultati dei comandi.

```
nome_script=‘basename $0‘
echo "Il nome di questo script è $nome_script."
```

Gli output dei comandi possono essere usati come argomenti per un altro comando, per impostare una variabile e anche per generare la lista degli argomenti in un ciclo for.

```
rm ‘cat nomefile‘   # “nomefile” contiene l’elenco dei file da cancellare.
#
# S. C. fa notare che potrebbe ritornare l’errore "arg list too long".
# Meglio              xargs rm -- < nomefile
# ( -- serve nei casi in cui “nomefile” inizia con un “-” )

elenco_filetesto=‘ls *.txt‘
#  La variabile contiene i nomi di tutti i file *.txt della directory
#+ di lavoro corrente.

echo $elenco_filetesto

elenco_filetesto2=$(ls *.txt)  # Forma alternativa di sostituzione di comando.
echo $elenco_filetesto2
# Stesso risultato.

# Un problema possibile, nell’inserire un elenco di file in un’unica stringa,
# è che si potrebbe insinuare un ritorno a capo.
#
#  Un modo più sicuro per assegnare un elenco di file ad un parametro
#+ è usare un array.
#      shopt -s nullglob       # Se non viene trovato niente, il nome del file
#+                               non viene espanso.
#      elenco_filetesto=( *.txt )
#
# Grazie, S.C.
```

Note: La sostituzione di comando invoca una subshell.

Caution

La sostituzione di comando può dar luogo alla suddivisione delle parole.

```
COMANDO ‘echo a b‘     # 2 argomenti: a e b;

COMANDO "‘echo a b‘"   # 1 argomento: "a b"

COMANDO ‘echo‘         # nessun argomento

COMANDO "‘echo‘"       # un argomento vuoto


# Grazie, S.C.
```

Anche quando la suddivisione delle parole non si verifica, la sostituzione di comando rimuove i ritorni a capo finali.

```
# cd "‘pwd‘"  # Questo dovrebbe funzionare sempre.
# Tuttavia...

mkdir ’directory con un carattere di a capo alla fine
’

cd ’directory con un carattere di a capo alla fine
’

cd "‘pwd‘"  # Messaggio d’errore:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # Funziona bene.


precedenti_impostazioni_tty=$(stty -g)  #  Salva le precedenti impostazioni
                                        #+ del terminale.
echo "Premi un tasto "
stty -icanon -echo                      #  Disabilita la modalità
                                        #+ "canonica" del terminale.
                                        #  Disabilita anche l’echo *locale*.

tasto=$(dd bs=1 count=1 2> /dev/null)   #  Uso di ’dd’ per rilevare il
                                        #+ tasto premuto.
stty "$precedenti_impostazioni_tty"     #  Ripristina le vecchie impostazioni.
echo "Hai premuto ${#tasto} tasto/i."   #  ${#variabile} = numero di caratteri
                                        #+ in $variabile
#
# Premete qualsiasi tasto tranne INVIO, l’output sarà "Hai premuto 1 tasto/i."
# Premete INVIO e sarà "Hai premuto 0 tasto/i."
# Nella sostituzione di comando i ritorni a capo vengono eliminati.

Grazie, S.C.
```

Caution

L'uso di **echo** per visualizzare una variabile *senza quoting*, e che è stata impostata con la sostituzione di comando, cancella i caratteri di ritorno a capo dall'output del/dei comando/i riassegnati. Questo può provocare spiacevoli sorprese.

```
elenco_directory=‘ls -l‘
echo $elenco_directory     # senza quoting

# Ci si potrebbe aspettare un elenco di directory ben ordinato.

# Invece, quello che si ottiene è:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# I ritorni a capo sono scomparsi.


echo "$elenco_directory"   # con il quoting
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

La sostituzione di comando consente anche l'impostazione di una variabile al contenuto di un file, sia usando la redirezione che con il comando cat.

```
variabile1=‘<file1‘      #  Imposta "variabile1" al contenuto di "file1".
variabile2=‘cat file2‘   #  Imposta "variabile2" al contenuto di "file2".
                         #  Questo, tuttavia, genera un nuovo processo, quindi
                         #+ questa riga di codice viene eseguita più
                         #+ lentamente della precedente.

#  Nota:
#  Le variabili potrebbero contenere degli spazi,
#+ o addirittura (orrore) caratteri di controllo.



#  Brani scelti dal file di sistema /etc/rc.d/rc.sysinit
#+ (su un’installazione Red Hat Linux)


if [ -f /fsckoptions ]; then
        fsckoptions=‘cat /fsckoptions‘
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; \
         then hdmedia=‘cat /proc/ide/${disk[$device]}/media‘
...
fi
```

```
#
#
if [ ! -n "‘uname -r | grep -- "-"‘" ]; then
       ktag="‘cat /proc/version‘"
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=‘cat /proc/bus/usb/devices \
                 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"‘
    kbdoutput=‘cat /proc/bus/usb/devices \
               2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"‘
...
fi
```

Caution

Non si imposti una variabile con il contenuto di un file di testo di *grandi dimensioni*, a meno che non si abbia una ragione veramente buona per farlo. Non si imposti una variabile con il contenuto di un file *binario*, neanche per scherzo.

Example 14-1. Stupid script tricks

```
#!/bin/bash
# stupid-script-tricks.sh: Gente, non eseguitelo!
# Da "Stupid Script Tricks," Volume I.


variabile_pericolosa=‘cat /boot/vmlinuz‘   # Il kernel Linux compresso.

echo "Lunghezza della stringa \$variabile_pericolosa = ${#variabile_pericolosa}"
# Lunghezza della stringa $variabile_pericolosa = 794151
# (Non dà lo stesso risultato di ’wc -c /boot/vmlinuz’.)

# echo "$variabile_pericolosa"
# Non fatelo, bloccherebbe l’esecuzione dello script.


#  L’autore di questo documento è consapevole dell’assoluta inutilità delle
#+ applicazioni che impostano una variabile al contenuto di un file binario.

exit 0
```

È da notare che in uno script non si verifica un *buffer overrun*. Questo è un esempio di come un linguaggio interpretato, qual’è Bash, possa fornire una maggiore protezione dagli errori del programmatore rispetto ad un linguaggio compilato.

La sostituzione di comando consente di impostare una variabile con l’output di un ciclo. La chiave per far ciò consiste nel racchiudere l’output del comando echo all’interno del ciclo.

Example 14-2. Generare una variabile da un ciclo

```
#!/bin/bash
# csubloop.sh: Impostazione di una variabile all’output di un ciclo.

variabile1=‘for i in 1 2 3 4 5
do
  echo -n "$i"                   #  Il comando ’echo’ è cruciale
done‘                            #+ nella sostituzione di comando.

echo "variabile1 = $variabile1"  # variabile1 = 12345


i=0
variabile2=‘while [ "$i" -lt 10 ]
do
  echo -n "$i"                   # Ancora, il necessario ’echo’.
  let "i += 1"                   # Incremento.
done‘

echo "variabile2 = $variabile2"  # variabile2 = 0123456789

exit 0
```

La sostituzione di comando permette di estendere la serie degli strumenti a disposizione di Bash. Si tratta semplicemente di scrivere un programma, o uno script, che invii il risultato allo `stdout` (come fa un ben funzionante strumento UNIX) e di assegnare quell'output ad una variabile.

```
#include <stdio.h>

/*  Programma C "Ciao, mondo"  */

int main()
{
  printf( "Ciao, mondo." );
  return (0);
}
```

```
bash$ gcc -o ciao ciao.c
```

```
#!/bin/bash
# hello.sh

saluti='./ciao'
echo $saluti
```

```
bash$ sh hello.sh
Ciao, mondo.
```

Note: Nella sostituzione di comando, la forma **$(COMANDO)** ha sostituito quella con gli apostrofi inversi.

```
output=$(sed -n /"$1"/p $file) # Dall'esempio "grp.sh".

# Impostazione di una variabile al contenuto di un file di testo.
Contenuto_file1=$(cat $file1)
Contenuto_file2=$(<$file2)     # Bash consente anche questa forma.
```

Esempi di sostituzione di comando negli script di shell:

1. Example 10-7
2. Example 10-26

3. Example 9-27
4. Example 12-2
5. Example 12-16
6. Example 12-13
7. Example 12-40
8. Example 10-13
9. Example 10-10
10. Example 12-25
11. Example 16-7
12. Example A-18
13. Example 28-1
14. Example 12-33
15. Example 12-34
16. Example 12-35

Notes

1. Nella *sostituzione di comando* il **comando** può essere rappresentato da un comando di sistema, da un *builtin di shell* o, addirittura, da una funzione dello script.

Chapter 15. Espansione aritmetica

L’espansione aritmetica fornisce un potente strumento per l’esecuzione delle operazioni matematiche negli script. È relativamente semplice trasformare una stringa in un’espressione numerica usando i costrutti apici inversi, doppie parentesi o let.

Variazioni

Espansione aritmetica con apici inversi (spesso usata in abbinamento con expr)

```
z=‘expr $z + 3‘          # Il comando ’expr’ esegue l’espansione.
```

Espansione aritmetica con doppie parentesi
ed uso di **let**

L’uso degli apici inversi nell’espansione aritmetica è stato sostituito dal costrutto doppie parentesi **`$((...))`** o dal convenientissimo **let**.

```
z=$(($z+3))
# $((ESPRESSIONE)) è l’espansione aritmetica. #  Non deve essere confusa
                                              #+ con la sostituzione di comando.

let z=z+3
let "z += 3"  #  Il quoting consente l’uso degli spazi e
              #+ degli operatori speciali.
#  L’operatore ’let’, in realtà, esegue una valutazione aritmetica,
#+ piuttosto che un’espansione.
```

Tutte le forme precedenti si equivalgono. È possibile usare quella che “vi è più consona”.

Esempi di espansione aritmetica negli script:

1. Example 12-7
2. Example 10-14
3. Example 26-1
4. Example 26-11
5. Example A-18

Chapter 16. Redirezione I/O

In modo predefinito, ci sono sempre tre “file” aperti: lo `stdin` (la tastiera), lo `stdout` (lo schermo) e lo `stderr` (la visualizzazione a schermo dei messaggi d’errore). Questi, e qualsiasi altro file aperto, possono essere rediretti. Redirezione significa semplicemente catturare l’output di un file, di un comando, di un programma, di uno script e persino di un blocco di codice presente in uno script (vedi Example 3-1 e Example 3-2), ed inviarlo come input ad un altro file, comando, programma o script.

Ad ogni file aperto viene assegnato un descrittore di file. [1] I descrittori di file per `stdin`, `stdout` e `stderr` sono, rispettivamente, 0, 1 e 2. Per aprire ulteriori file rimangono i descrittori dal 3 al 9. Talvolta è utile assegnare uno di questi descrittori di file addizionali allo `stdin`, `stdout` o `stderr` come duplicato temporaneo. [2] Questo semplifica il ripristino ai valori normali dopo una redirezione complessa (vedi Example 16-1).

```
   OUTPUT_COMANDO >
      # Redirige lo stdout in un file.
      # Crea il file se non è presente, in caso contrario lo sovrascrive.

      ls -lR > dir-albero.list
      # Crea un file contente l’elenco dell’albero delle directory.

   : > nomefile
      # Il > svuota il file "nomefile", lo riduce a dimensione zero.
      # Se il file non è presente, ne crea uno vuoto (come con ’touch’).
      # I : servono da segnaposto e non producono alcun output.

    > nomefile
      #  Il > svuota il file "nomefile", lo riduce a dimensione zero.
      #  Se il file non è presente, ne crea uno vuoto (come con ’touch’).
      #  (Stesso risultato di ": >", visto prima, ma questo, con alcune
      #+ shell, non funziona.)

   OUTPUT_COMANDO >>
      # Redirige lo stdout in un file.
      # Crea il file se non è presente, in caso contrario accoda l’output.


      #  Comandi di redirezione di riga singola
      #+ (hanno effetto solo sulla riga in cui si trovano):
--------------------------------------------------------------------

   1>nomefile
      # Redirige lo stdout nel file "nomefile".
    1>>nomefile
      # Redirige e accoda lo stdout nel file "nomefile".
    2>nomefile
      # Redirige lo stderr nel file "nomefile".
    2>>nomefile
      # Redirige e accoda lo stderr nel file "nomefile".
    &>nomefile
      # Redirige sia lo stdout che lo stderr nel file "nomefile".
```

```
#============================================================================
      # Redirigere lo stdout una riga alla volta.
      FILELOG=script.log

      echo "Questo enunciato viene inviato al file di log, \
           \"$FILELOG\"." 1>$FILELOG
      echo "Questo enunciato viene accodato in \"$FILELOG\"." 1>>$FILELOG
      echo "Anche questo enunciato viene accodato in \"$FILELOG\"." 1>>$FILELOG
      echo "Questo enunciato viene visualizzato allo \
            stdout e non comparirà in \"$FILELOG\"."
      #  Questi comandi di redirezione vengono automaticamente "annullati"
      #+ dopo ogni riga.



      # Redirigere lo stderr una riga alla volta.
      FILEERRORI=script.err

      comando_errato1 2>$FILEERRORI          #  Il messaggio d'errore viene
                                             #+ inviato in $FILEERRORI.
      comando_errato2 2>>$FILEERRORI         #  Il messaggio d'errore viene
                                             #+ accodato in $FILEERRORI.
      comando_errato3                        #  Il messaggio d'errore viene
                                             #+ visualizzato allo stderr,
                                             #+ e non comparirà in $FILEERRORI.
      #  Questi comandi di redirezione vengono automaticamente "annullati"
      #+ dopo ogni riga.

#============================================================================



   2>&1
      # Redirige lo stderr allo stdout.
      # I messaggi d'errore vengono visualizzati a video, ma come stdout.

    i>&j
      #  Redirige il descrittore di file i in j.
      #  Tutti gli output del file puntato da i vengono inviati al file
      #+ puntato da j.

   >&j
      # Redirige, per default, il descrittore di file 1 (lo stdout) in j.
      # Tutti gli stdout vengono inviati al file puntato da j.

    0< NOMEFILE
     < NOMEFILE
      # Riceve l'input da un file.
      # È il compagno di ">" e vengono spesso usati insieme.
      #
      # grep parola-da-cercare <nomefile
```

```
[j]<>nomefile
   #  Apre il file "nomefile" in lettura e scrittura, e gli assegna il
   #+ descrittore di file "j".
   #  Se il file "nomefile" non esiste, lo crea.
   #  Se il descrittore di file "j" non viene specificato, si assume per
   #+ default il df 0, lo stdin.
   #
   #  Una sua applicazione è quella di scrivere in un punto specifico
   #+ all'interno di un file.
   echo 1234567890 > File    #  Scrive la stringa in "File".
   exec 3<> File             #  Apre "File" e gli assegna il df 3.
   read -n 4 <&3             #  Legge solo 4 caratteri.
   echo -n . >&3             #  Scrive il punto decimale dopo i
                             #+ caratteri letti.
   exec 3>&-                 #  Chiude il df 3.
   cat File                  #  ==> 1234.67890
   # È l'accesso casuale, diamine.

|
   # Pipe.
   # Strumento generico per concatenare processi e comandi.
   # Simile a ">", ma con effetti più generali.
   # Utile per concatenare comandi, script, file e programmi.
   cat *.txt | sort | uniq > file-finale
   # Ordina gli output di tutti i file .txt e cancella le righe doppie,
   # infine salva il risultato in "file-finale".
```

È possibile combinare molteplici istanze di redirezione input e output e/o pipe in un'unica linea di comando.

```
comando < file-input > file-output

comando1 | comando2 | comando3 > file-output
```

Vedi Example 12-24 e Example A-16.

È possibile redirigere più flussi di output in un unico file.

```
ls -yz >> comandi.log 2>&1
# Invia il risultato delle opzioni illegali "yz" di "ls" nel file "comandi.log"
#  Poiché lo stderr è stato rediretto nel file, in esso si troveranno anche
#+ tutti i messaggi d'errore.
```

Chiusura dei descrittori di file

n<&-

Chiude il descrittore del file di input *n*.

0<&-
<&-

Chiude lo `stdin`.

n>&-

Chiude il descrittore del file di output *n*.

1>&-
>&-

Chiude lo `stdout`.

I processi figli ereditano dai processi genitore i descrittori dei file aperti. Questo è il motivo per cui le pipe funzionano. Per evitare che un descrittore di file venga ereditato è necessario chiuderlo.

```
# Redirige solo lo stderr alla pipe.

exec 3>&1                           #  Salva il "valore" corrente dello stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&- #  Chiude il df 3 per 'grep' (ma
                                    #+ non per'ls').
#              ^^^^   ^^^^
exec 3>&-                           #  Ora è chiuso anche per la parte
                                    #+ restante dello script.

# Grazie, S.C.
```

Per una più dettagliata introduzione alla redirezione I/O vedi Appendix E.

16.1. Usare exec

Il comando **exec <nomefile** redirige lo `stdin` nel file indicato. Da quel punto in avanti tutto lo `stdin` proverrà da quel file, invece che dalla normale origine (solitamente l'input da tastiera). In questo modo si dispone di un mezzo per leggere un file riga per riga con la possibilità di verificare ogni riga di input usando sed e/o awk.

Example 16-1. Redirigere lo `stdin` usando exec

```
#!/bin/bash
# Redirigere lo stdin usando 'exec'.


exec 6<&0          # Collega il descrittore di file nr.6 allo stdin.
                   # Salva lo stdin.

exec < file-dati   # lo stdin viene sostituito dal file "file-dati"

read a1            # Legge la prima riga del file "file-dati".
read a2            # Legge la seconda riga del file "file-dati."

echo
echo "Le righe lette dal file."
```

```
echo "------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  È stato ripristinato lo stdin dal df nr.6, dov'era stato salvato,
#+ e chiuso il df nr.6 ( 6<&- ) per renderlo disponibile per un altro processo.
#
# <&6 6<&-    anche questo va bene.

echo -n "Immetti dei dati  "
read b1  # Ora "read" funziona come al solito, leggendo dallo stdin.
echo "Input letto dallo stdin."
echo "------------------------"
echo "b1 = $b1"

echo

exit 0
```

In modo simile, il comando **exec >nomefile** redirige lo `stdout` nel file indicato. In questo modo, tutti i risultati dei comandi, che normalmente verrebbero visualizzati allo `stdout`, vengono inviati in quel file.

Example 16-2. Redirigere lo `stdout` utilizzando exec

```
#!/bin/bash
# reassign-stdout.sh

FILELOG=filelog.txt

exec 6>&1           # Collega il descrittore di file nr.6 allo stdout.
                    # Salva lo stdout.

exec > $FILELOG     # stdout sostituito dal file "filelog.txt".

# ----------------------------------------------------------------------- #
#  Tutti i risultati dei comandi inclusi in questo blocco di codice vengono
#+ inviati al file $FILELOG.

echo -n "File di log: "
date
echo "-------------------------------------"
echo

echo "Output del comando \"ls -al\""
echo
ls -al
echo; echo
echo "Output del comando \"df\""
echo
df
```

```
# ------------------------------------------------------------------- #

exec 1>&6 6>&-    # Ripristina lo stdout e chiude il descrittore di file nr.6.

echo
echo "== ripristinato lo stdout alla funzionalità di default  == "
echo
ls -al
echo


exit 0
```

Example 16-3. Redirigere, nello stesso script, sia lo `stdin` che lo `stdout` con exec

```
#!/bin/bash
# upperconv.sh
# Converte in lettere maiuscole il testo del file di input specificato.

E_ACCESSO_FILE=70
E_ERR_ARG=71

if [ ! -r "$1" ]     # Il file specificato ha i permessi in lettura?
then
  echo "Non riesco a leggere il file di input!"
  echo "Utilizzo: $0 file-input file-output"
  exit $E_ACCESSO_FILE
fi                   #  Esce con lo stesso errore anche
                     #+ quando non viene specificato il file di input ($1).

if [ -z "$2" ]
then
  echo "Occorre specificare un file di output."
  echo "Utilizzo: $0 file-input file-output"
  exit $E_ERR_ARG
fi


exec 4<&0
exec < $1            # Per leggere dal file di input.


exec 7>&1
exec > $2            #  Per scrivere nel file di output.
                     #  Nell'ipotesi che il file di output abbia i permessi
                     #+ di scrittura (aggiungiamo una verifica?).

# ------------------------------------------------------
    cat - | tr a-z A-Z   # Conversione in lettere maiuscole.
#   ^^^^^               # Legge dallo stdin.
#          ^^^^^^^^^^   # Scrive sullo stdout.
```

```
# Comunque, sono stati rediretti sia lo stdin che lo stdout.
# ----------------------------------------------------------

exec 1>&7 7>&-       # Ripristina lo stdout.
exec 0<&4 4<&-       # Ripristina lo stdin.

# Dopo il ripristino, la riga seguente viene visualizzata allo stdout
#+ come ci aspettiamo.
echo "Il file \"$1\" è stato scritto in \"$2\" in lettere maiuscole."


exit 0
```

16.2. Redirigere blocchi di codice

Anche i blocchi di codice, come i cicli while, until e for, nonché i costrutti di verifica if/then, possono prevedere la redirezione dello `stdin`. Persino una funzione può usare questa forma di redirezione (vedi Example 23-7). L'operatore <, posto alla fine del blocco, esegue questo compito.

Example 16-4. Ciclo *while* rediretto

```
#!/bin/bash

if [ -z "$1" ]
then
  Nomefile=nomi.data        #  È il file predefinito, se non ne viene
                            #+ specificato alcuno.
else
  Nomefile=$1
fi
#+ Nomefile=${1:-nomi.data}
#  può sostituire la verifica precedente (sostituzione di parametro).

conto=0

echo

while [ "$nome" != Smith ]  #  Perché la variabile $nome è stata usata
                            #+ con il quoting?
do
  read nome                 # Legge da $Nomefile invece che dallo stdin.
  echo $nome
  let "conto += 1"
done <"$Nomefile"           # Redirige lo stdin nel file $Nomefile.
#    ^^^^^^^^^^^

echo; echo "$conto nomi letti"; echo

#  È da notare che, in alcuni linguaggi di scripting di shell più vecchi, il
#+ ciclo rediretto viene eseguito come una subshell.
#  Di conseguenza, $conto restituirebbe 0, il valore di inizializzazione prima
```

```
#+ del ciclo.
#  Bash e ksh evitano l'esecuzione di una subshell ogni volta che è possibile,
#+ cosicché questo script, ad esempio, funziona correttamente.
#
# Grazie a Heiner Steven per la precisazione.


exit 0
```

Example 16-5. Una forma alternativa di ciclo *while* rediretto

```
#!/bin/bash

# Questa è una forma alternativa dello script precedente.

#  Suggerito da Heiner Steven
#+ come espediente in quelle situazioni in cui un ciclo rediretto
#+ viene eseguito come subshell e, quindi, le variabili all'interno del ciclo
#+ non conservano i loro valori dopo che lo stesso è terminato.


if [ -z "$1" ]
then
  Nomefile=nomi.data      #  È il file predefinito, se non ne viene
                          #+ specificato alcuno.
else
  Nomefile=$1
fi


exec 3<&0                 # Salva lo stdin nel descrittore di file 3.
exec 0<"$Nomefile"        # Redirige lo standard input.

conto=0
echo


while [ "$nome" != Smith ]
do
  read nome               # Legge dallo stdin rediretto ($Nomefile).
  echo $nome
  let "conto += 1"
done <"$Nomefile"         # Il ciclo legge dal file $Nomefile.
#    ^^^^^^^^^^^^


exec 0<&3                 # Ripristina il precedente stdin.
exec 3<&-                 # Chiude il temporaneo df 3.

echo; echo "$conto nomi letti"; echo

exit 0
```

Example 16-6. Ciclo *until* rediretto

```
#!/bin/bash
# Uguale all'esempio precedente, ma con il ciclo "until".

if [ -z "$1" ]
then
  Nomefile=nomi.data          #  È il file predefinito, se non ne viene
                              #+ specificato alcuno.
else
  Nomefile=$1
fi

# while [ "$nome" != Smith ]
until [ "$nome" = Smith ]     # Il != è cambiato in =.
do
  read nome                   # Legge da $Nomefile, invece che dallo stdin.
  echo $nome
done <"$Nomefile"             # Redirige lo stdin nel file $Nomefile.
#    ^^^^^^^^^^^^

# Stessi risultati del ciclo "while" dell'esempio precedente.

exit 0
```

Example 16-7. Ciclo *for* rediretto

```
#!/bin/bash

if [ -z "$1" ]
then
  Nomefile=nomi.data           #  È il file predefinito, se non ne viene
                               #+ specificato alcuno.
else
  Nomefile=$1
fi

conta_righe=`wc $Nomefile | awk '{ print $1 }'`
#           Numero di righe del file indicato.
#
#  Elaborato e con diversi espedienti, ciò nonostante mostra che
#+ è possibile redirigere lo stdin in un ciclo "for" ...
#+ se siete abbastanza abili.
#
# Più conciso     conta_righe=$(wc < "$Nomefile")


for nome in `seq $conta_righe` #  Ricordo che "seq" genera una sequenza
       #+ di numeri.
# while [ "$nome" != Smith ]   --  più complicato di un ciclo "while"  --
do
  read nome                   #  Legge da $Nomefile, invece che dallo stdin.
```

```
  echo $nome
  if [ "$nome" = Smith ]       #  Sono necessarie tutte queste istruzioni
                               #+ aggiuntive.
  then
      break
  fi
done <"$Nomefile"              #  Redirige lo stdin nel file $Nomefile.
#    ^^^^^^^^^^^^

exit 0
```

Il precedente esempio può essere modificato per redirigere anche l'output del ciclo.

Example 16-8. Ciclo *for* rediretto (rediretti sia lo `stdin` che lo `stdout`)

```
#!/bin/bash

if [ -z "$1" ]
then
  Nomefile=nomi.data             #  È il file predefinito, se non ne viene
                                 #+ specificato alcuno.
else
  Nomefile=$1
fi

Filereg=$Nomefile.nuovo          #  Nome del file in cui vengono salvati i
                                 #+ risultati.
NomeFinale=Jonah                 #  Nome per terminare la "lettura".

conta_righe=‘wc $Nomefile | awk ’{ print $1 }’‘ #  Numero di righe del file
                                                #+ indicato.

for nome in ‘seq $conta_righe‘
do
  read nome
  echo "$nome"
  if [ "$nome" = "$NomeFinale" ]
  then
      break
  fi
done < "$Nomefile" > "$Filereg"  # Redirige lo stdin nel file $Nomefile,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^    e lo salva nel file di backup $Filereg.

exit 0
```

Example 16-9. Costrutto *if/then* rediretto

```
#!/bin/bash

if [ -z "$1" ]
then
  Nomefile=nomi.data    #  È il file predefinito, se non ne viene
```

```
                        #+ specificato alcuno.
else
  Nomefile=$1
fi

TRUE=1

if [ "$TRUE" ]          # vanno bene anche  if true   e   if :
then
 read nome
 echo $nome
fi <"$Nomefile"
#  ^^^^^^^^^^^^

#  Legge solo la prima riga del file.
#  Il costrutto "if/then" non possiede alcuna modalità di iterazione
#+ se non inserendolo in un ciclo.

exit 0
```

Example 16-10. File dati “nomi.data” usato negli esempi precedenti

```
Aristotile
Belisario
Capablanca
Eulero
Goethe
Hamurabi
Jonah
Laplace
Maroczy
Purcell
Schmidt
Semmelweiss
Smith
Turing
Venn
Wilson
Znosko-Borowski

#  Questo è il file dati per
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Redirigere lo `stdout` di un blocco di codice ha l’effetto di salvare il suo output in un file. Vedi Example 3-2.

Gli here document rappresentano casi particolari di blocchi di codice rediretti.

16.3. Applicazioni

Un uso intelligente della redirezione I/O consente di mettere insieme, e verificare, frammenti di output dei comandi (vedi Example 11-6). Questo permette di generare dei rapporti e dei file di log.

Example 16-11. Eventi da registrare in un file di log

```
#!/bin/bash
# logevents.sh, di Stephane Chazelas.

# Evento da registrare in un file.
# Deve essere eseguito da root (per l'accesso in scrittura a /var/log).

ROOT_UID=0     # Solo gli utenti con $UID 0 hanno i privilegi di root.
E_NONROOT=67   # Errore di uscita non root.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Bisogna essere root per eseguire lo script."
  exit $E_NONROOT
fi


DF_DEBUG1=3
DF_DEBUG2=4
DF_DEBUG3=5

# Decommentate una delle due righe seguenti per attivare lo script.
# LOG_EVENTI=1
# LOG_VAR=1


log()  # Scrive la data e l'ora nel file di log.
{
echo "$(date)  $*" >&7     # *Accoda* la data e l'ora nel file.
                           # Vedi oltre.
}



case $LIVELLO_LOG in
 1) exec 3>&2         4> /dev/null 5> /dev/null;;
 2) exec 3>&2         4>&2         5> /dev/null;;
 3) exec 3>&2         4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

DF_LOGVAR=6
if [[ $LOG_VAR ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null               # Sopprime l'output.
fi
```

```
DF_LOGEVENTI=7
if [[ $LOG_EVENTI ]]
then
  # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # La riga precedente non funziona nella versione Bash 2.04.
  exec 7>> /var/log/event.log        # Accoda in "event.log".
  log                                # Scrive la data e l'ora.
else exec 7> /dev/null               # Sopprime l'output.
fi

echo "DEBUG3: inizio" >&${DF_DEBUG3}

ls -l >&5 2>&4                       # comando1 >&5 2>&4

echo "Fatto"                         # comando2

echo "invio mail" >&${DF_LOGEVENTI}  # Scrive "invio mail" nel df nr.7.


exit 0
```

Notes

1. Un *descrittore di file* è semplicemente un numero che il sistema operativo assegna ad un file aperto per tenerne traccia. Lo si consideri una versione semplificata di un puntatore ad un file. È analogo ad un *gestore di file* del C.
2. L'uso del *`descrittore di file`* 5 potrebbe causare problemi. Quando Bash crea un processo figlio, come con exec, il figlio eredita il df 5 (vedi la e-mail di Chet Ramey in archivio, SUBJECT: RE: File descriptor 5 is held open (http://www.geocrawler.com/archives/3/342/1996/1/0/1939805/)). Meglio non utilizzare questo particolare descrittore di file.

Chapter 17. Here document

Un *here document* è un blocco di codice con una funzione specifica e che utilizza una particolare forma di redirezione I/O per fornire un elenco di comandi a programmi o comandi interattivi, come ftp, telnet, o **ex**. Una "stringa limite" delimita (incornicia) l'elenco dei comandi. Il simbolo speciale << indica la stringa limite. Questo ha come effetto la redirezione dell'output di un file nello `stdin` di un programma o di un comando. È simile a **`programma-interattivo < file-comandi`**, dove **`file-comandi`** contiene

```
comando nr.1
comando nr.2
...
```

L'alternativa rappresentata da un "here document" è la seguente:

```
#!/bin/bash
programma-interattivo <<StringaLimite
comando nr.1
comando nr.2
...
StringaLimite
```

Si scelga, per la stringa limite, un nome abbastanza insolito in modo che non ci sia la possibilità che lo stesso nome compaia accidentalmente nell'elenco dei comandi e causare confusione.

Si noti che gli *here document* possono, talvolta, essere usati efficacemente anche con utility e comandi non interattivi.

Example 17-1. File di prova: Crea un file di prova di 2 righe

```
#!/bin/bash

# Uso non interattivo di 'vi' per redigere un file.
# Emula 'sed'.

E_ERR_ARG=65

if [ -z "$1" ]
then
  echo "Utilizzo: 'basename $0' nomefile"
  exit $E_ERR_ARG
fi

FILE=$1

# Inserisce 2 righe nel file, quindi lo salva.
#--------Inizio here document----------------#
vi $FILE <<x23StringaLimitex23
i
Questa è la riga 1 del file d'esempio.
Questa è la riga 2 del file d'esempio.
```

```
^[
ZZ
x23StringaLimitex23
#----------Fine here document---------------#

#  Notate che i caratteri ^[ corrispondono alla
#+ digitazione di Control-V <Esc>.

#  Bram Moolenaar fa rilevare che questo potrebbe non funzionare con 'vim',
#+ a causa di possibili problemi di interazione con il terminale.

exit 0
```

Lo script precedente si potrebbe, semplicemente ed efficacemente, implementare con **ex**, invece che con **vi**. Gli here document che contengono una lista di comandi **ex** sono abbastanza comuni e formano una specifica categoria a parte, conosciuta come *ex script*.

Example 17-2. Trasmissione: Invia un messaggio a tutti gli utenti connessi

```
#!/bin/bash

wall <<zzz23FineMessaggiozzz23
Inviate per e-mail gli ordini per la pizza di mezzogiorno
all'amministratore di  sistema.
    (Aggiungete un euro extra se la volete con acciughe o funghi.)
# Il testo aggiuntivo del messaggio va inserito qui.
# Nota: 'wall' visualizza le righe di commento.
zzz23FIneMessaggiozzz23

# Si sarebbe potuto fare in modo più efficiente con
#         wall <file-messaggio
# Comunque, salvare un messaggio campione in uno script risparmia del lavoro.

exit 0
```

Example 17-3. Messaggio di più righe usando cat

```
#!/bin/bash

#  'echo' è ottimo per visualizzare messaggi di una sola riga,
#+ ma diventa problematico per messaggi più lunghi.
#  Un here document 'cat' supera questa limitazione.

cat <<Fine-messaggio
-------------------------------------
Questa è la riga 1 del messaggio.
Questa è la riga 2 del messaggio.
Questa è la riga 3 del messaggio.
Questa è la riga 4 del messaggio.
Questa è l'ultima riga del messaggio.
-------------------------------------
Fine-messaggio
```

```
exit 0


#-----------------------------------------
# Il codice che segue non viene eseguito per l'"exit 0" precedente.

# S.C. sottolinea che anche la forma seguente funziona.
echo "-----------------------------------
Questa è la riga 1 del messaggio.
Questa è la riga 2 del messaggio.
Questa è la riga 3 del messaggio.
Questa è la riga 4 del messaggio.
Questa è l'ultima riga del messaggio.
-----------------------------------------"
#  Tuttavia, il testo non dovrebbe contenere  doppi apici senza il
#+ carattere di escape.
```

L'opzione `-` alla stringa limite del here document (**`<<-StringaLimite`**) sopprime i caratteri di tabulazione (ma non gli spazi) nell'output. Può essere utile per rendere lo script più leggibile.

Example 17-4. Messaggio di più righe con cancellazione dei caratteri di tabulazione

```
#!/bin/bash
# Uguale all'esempio precedente, ma...

#  L'opzione - al here document <<-
#+ sopprime le tabulazioni nel corpo del documento, ma *non* gli spazi.

cat <<-FINEMESSAGGIO
        Questa è la riga 1 del messaggio.
        Questa è la riga 2 del messaggio.
        Questa è la riga 3 del messaggio.
        Questa è la riga 4 del messaggio.
        Questa è l'ultima riga del messaggio.
FINEMESSAGGIO
# L'output dello script viene spostato a sinistra.
# Le tabulazioni iniziali di ogni riga non vengono mostrate.

#  Le precedenti 5 righe del "messaggio" sono precedute da
#+ tabulazioni, non da spazi.
#  Gli spazi non sono interessati da  <<-.


exit 0
```

Un here document supporta la sostituzione di comando e di parametro. È quindi possibile passare diversi parametri al corpo del here document e modificare, conseguentemente, il suo output.

Example 17-5. Here document con sostituzione di parametro

```
#!/bin/bash
# Un altro here document 'cat' che usa la sostituzione di parametro.

# Provatelo senza nessun parametro da riga di comando,   ./nomescript
# Provatelo con un parametro da riga di comando,   ./nomescript Mortimer
# Provatelo con un parametro di due parole racchiuse tra doppi apici,
#                           ./nomescript "Mortimer Jones"

LINEACMDPARAM=1    # Si aspetta almeno un parametro da linea di comando.

if [ $# -ge $LINEACMDPARAM ]
then
  NOME=$1          # Se vi è più di un parametro,
                   # tiene conto solo del primo.
else
  NOME="John Doe"  # È il nome predefinito, se non si passa alcun parametro.
fi

RISPONDENTE="l'autore di questo bello script"


cat <<Finemessaggio

Ciao, sono $NOME.
Salute a te $NOME, $RISPONDENTE.

# Questo commento viene visualizzato nell'output (perché?).

Finemessaggio

# Notate che vengono visualizzate nell'output anche le righe vuote.
# Così si fa un "commento".

exit 0
```

Quello che segue è un utile script contenente un here document con sostituzione di parametro.

Example 17-6. Caricare due file nella directory incoming di "Sunsite"

```
#!/bin/bash
# upload.sh

#  Carica due file (Nomefile.lsm, Nomefile.tar.gz)
#+ nella directory incoming di Sunsite/UNC (ibiblio.org).
#  Nomefile.tar.gz è l'archivio vero e proprio.
#  Nomefile.lsm è il file descrittore.


E_ERR_ARG=65

if [ -z "$1" ]
```

```
then
  echo "Utilizzo: ‘basename $0‘ nomefile-da-caricare"
  exit $E_ERR_ARG
fi


Nomefile=‘basename $1‘              # Toglie il percorso dal nome del file.

Server="ibiblio.org"
Directory="/incoming/Linux"
#  Questi dati non dovrebbero essere codificati nello script,
#+ ma si dovrebbe avere la possibilità di cambiarli fornendoli come
#+ argomenti da riga di comando.

Password="vostro.indirizzo.e-mail"  # Sostituitelo con quello appropriato.

ftp -n $Server <<Fine-Sessione
# l’opzione -n disabilita l’auto-logon

user anonymous "$Password"
binary
bell                                #  Emette un ’segnale acustico’ dopo ogni
                                    #+ trasferimento di file
cd $Directory
put "$Nomefile.lsm"
put "$Nomefile.tar.gz"
bye
Fine-Sessione

exit 0
```

L’uso del quoting o dell’escaping sulla “stringa limite” del here document disabilita la sostituzione di parametro all’interno del suo corpo.

Example 17-7. Sostituzione di parametro disabilitata

```
#!/bin/bash
#  Un here document ’cat’ con la sostituzione di parametro disabilitata.

NOME="John Doe"
RISPONDENTE="L’autore dello script"

cat <<’Finemessaggio’

Ciao, sono $NOME.
Salute a te $NOME, $RISPONDENTE.

Finemessaggio

#  Non c’è sostituzione di parametro quando si usa il quoting o l’escaping
#+ sulla "stringa limite".
#  Le seguenti notazioni avrebbero avuto, entrambe, lo stesso effetto.
#  cat <"Finemessaggio"
```

```
#  cat <\Finemessaggio

exit 0
```

Disabilitare la sostituzione di parametro permette la produzione di un testo letterale. Questo può essere sfruttato per generare degli script o perfino il codice di un programma.

Example 17-8. Uno script che genera un altro script

```
#!/bin/bash
# generate-script.sh
# Basato su un'idea di Albert Reiner.

OUTFILE=generato.sh          # Nome del file da generare.


# ----------------------------------------------------------
# 'Here document contenente il corpo dello script generato.
(
cat <<'EOF'
#!/bin/bash

echo "Questo è uno script di shell generato (da un altro script)."
#  È da notare che, dal momento che ci troviamo all'interno di una subshell,
#+ non possiamo accedere alle variabili dello script "esterno".
#  C'è, solo da provare . . .
echo "Il file prodotto si chiamerà: $OUTFILE"  # Non funziona.

a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $OUTFILE
# ----------------------------------------------------------

#  Il quoting della 'stringa limite' evita l'espansione di variabile
#+ all'interno del corpo del precedente 'here document.'
#  Questo consente di conservare le stringhe letterali nel file prodotto.

if [ -f "$OUTFILE" ]
then
  chmod 755 $OUTFILE
  # Rende eseguibile il file generato.
else
  echo "Problema nella creazione del file: \"$OUTFILE\""
fi

#  Questo metodo può anche essere usato per generare
#+ programmi C, Perl, Python, Makefiles e simili.
```

```
exit 0
```

Un here document può fornire l’input ad una funzione del medesimo script.

Example 17-9. Here document e funzioni

```
#!/bin/bash
# here-function.sh

AcquisisceDatiPersonali ()
{
  read nome
  read cognome
  read indirizzo
  read città
  read cap
  read nazione
} # Può certamente apparire come una funzione interattiva, ma...


# Forniamo l’input alla precedente funzione.
AcquisisceDatiPersonali <<RECORD001
Ferdinando
Rossi
Via XX Settembre, 69
Milano
20100
ITALIA
RECORD001


echo
echo "$nome $cognome"
echo "$indirizzo"
echo "$città,  $cap, $nazione"
echo


exit 0
```

È possibile usare i : come comando fittizio per ottenere l’output di un here document. Si crea, così, un here document “anonimo”.

Example 17-10. Here document “anonimo”

```
#!/bin/bash

: <<VERIFICAVARIABILI
${HOSTNAME?}${USER?}${MAIL?}  #  Visualizza un messaggio d’errore se una
                              #+ delle variabili non è impostata.
VERIFICAVARIABILI
```

```
exit 0
```

Tip: Una variazione della precedente tecnica consente di “commentare” blocchi di codice.

Example 17-11. Commentare un blocco di codice

```
#!/bin/bash
# commentblock.sh

: << BLOCCOCOMMENTO
echo "Questa riga non viene visualizzata."
Questa è una riga di commento senza il carattere "#"
Questa è un’altra riga di commento senza il carattere "#"

&*@!!++=
La riga precedente non causa alcun messaggio d’errore,
perché l’interprete Bash la ignora.
BLOCCOCOMMENTO

echo "Il valore di uscita del precedente \"BLOCCOCOMMENTO\" è $?."   # 0
# Non viene visualizzato alcun errore.


#  La tecnica appena mostrata diventa utile anche per commentare
#+ un blocco di codice a scopo di debugging.
#  Questo evita di dover mettere il "#" all’inizio di ogni riga,
#+ e quindi di dovere, più tardi, ricominciare da capo e cancellare tutti i "#"

: << DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX

exit 0
```

Tip: Un’altra variazione di questo efficace espediente rende possibile l’“auto-documentazione” degli script.

Example 17-12. Uno script che si auto-documenta

```
#!/bin/bash
# self-document.sh: script autoesplicativo
# È una modifica di "colm.sh".

RICHIESTA_DOCUMENTAZIONE=70
```

```
if [ "$1" = "-h"  -o "$1" = "--help" ]     # Richiesta d'aiuto.
then
  echo; echo "Utilizzo: $0 [nome-directory]"; echo
  sed --silent -e '/DOCUMENTAZIONEXX$/,/^DOCUMENTAZIONE/p' "$0" |
  sed -e '/DOCUMENTAZIONEXX/d'; exit $RICHIESTA_DOCUMENTAZIONE; fi

: << DOCUMENTAZIONEXX
Elenca le statistiche di una directory specificata in formato tabellare.
---------------------------------------------------------------------------
Il parametro da riga di comando specifica la directory  di cui si desiderano
le statistiche. Se non è specificata alcuna directory o quella indicata non
può essere letta, allora vengono visualizzate le statistiche della directory
di lavoro corrente.

DOCUMENTAZIONEXX

if [ -z "$1" -o ! -r "$1" ]
then
  directory=.
else
  directory="$1"
fi

echo "Statistiche di "$directory":"; echo
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l "$directory" | sed 1d) | column -t

exit 0
```

Note: Gli here document creano file temporanei che vengono cancellati subito dopo la loro apertura, e non sono accessibili da nessun altro processo.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh (deleted)
```

Caution

Alcune utility non funzionano se inserite in un *here document*.

Warning

La *stringa limite* di chiusura, dopo l'ultima riga di un here document, deve iniziare esattamente dalla *prima* posizione della riga. Non deve esserci *nessuno spazio iniziale*.

```
#!/bin/bash

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

cat <<StringaLimite
echo "Questa è la riga 1 del messaggio contenuto nel here document."
echo "Questa è la riga 2 del messaggio contenuto nel here document."
echo "Questa è la riga finale del messaggio contenuto nel here document."
     StringaLimite
#^^^^String limite indentata. Errore! Lo script non si comporta come speravamo.

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

#  Questi commenti si trovano esternamente al 'here document'
#+ e non vengono visualizzati.

echo "Fuori dal here document."

exit 0

echo "Questa riga sarebbe meglio evitarla."  # Viene dopo il comando 'exit'.
```

Per quei compiti che risultassero troppo complessi per un "here document", si prenda in considerazione l'impiego del linguaggio di scripting **expect** che è particolarmente adatto a fornire gli input ai programmi interattivi.

Chapter 18. Intervallo

Questa bizzarra e breve intromissione dà al lettore
la possibilità di rilassarsi e, forse, di sorridere un po'.

Compagno utilizzatore di Linux, salve! Stai leggendo qualcosa che ti porterà fortuna e buona sorte. Spedisci semplicemente per e-mail una copia di questo documento a dieci tuoi amici. Però, prima di fare le copie, invia uno script Bash di cento righe alla prima persona dell'elenco che trovi allegato in fondo alla lettera. Quindi cancella il suo nome ed aggiungi il tuo alla fine della lista stessa.

Non spezzare la catena! Fai le copie entro quarantott'ore. Wilfred P. di Brooklyn non ha spedito le sue dieci copie e si è svegliato, il mattino dopo, scoprendo che la sua qualifica di lavoro era cambiata in "programmatore COBOL." Howard L. di Newport News ha spedito le copie ed in un mese ha avuto abbastanza hardware per costruirsi un cluster Beowulf a cento nodi che ha riservato per giocare a xbill. Amelia V. di Chicago ha riso leggendo questa lettera e ha interrotto la catena. Poco dopo il suo terminale ha preso fuoco e ora lei passa i suoi giorni scrivendo documentazione per MS Windows.

Non spezzare la catena! Spedisci le copie oggi stesso!

Cortesia di 'NIX "fortune cookies", con qualche modifica e molti ringraziamenti

Part 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?).

Chapter 19. Espressioni Regolari

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 che UNIX dota di 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 qualsiasi carattere, 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.

  > **Note:** 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.
```

- **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).

```
# La versione GNU di sed e awk può usare "+",
# ma è necessario l'escaping.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# Tutte queste forme si equivalgono.

# 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).

  > **Note:** 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.
```

> **Note:** Alcune versioni di **sed**, **ed** e **ex** supportano le versioni con escaping delle espressioni regolari estese descritte prima.

- **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]`**.

  Important: 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'Example 12-15 e Example 12-16.

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

"Sed & Awk", di Dougherty e Robbins fornisce una trattazione esaustiva e lucida delle ER (vedi *Bibliography*).

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 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`. [3] 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$ echo *
a.1 b.1 c.1 t2.sh test1.txt

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

Anche il comando echo esegue l'espansione dei caratteri jolly nei nomi dei file.

> **Note:** 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 Example 10-4.

Notes

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.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here document
riga1
riga2
EOF
# OUTPUT:
# [riga1
# riga2]



echo

awk '{ $0=$1 "\n" $2; if (/riga.1/) {print}}' << EOF
riga 1
riga 2
EOF
# OUTPUT:
# riga
# 1


# Grazie, S.C.

exit 0
```

3. 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.

```
~/[.]bashrc    #  Non viene espanso a ~/.bashrc
~/?bashrc      #  Neanche in questo caso.
               #  Nel globbing i caratteri jolly e i metacaratteri non
               #+ espandono il punto.

~/.[b]ashrc    #  Viene espanso a ~./bashrc
~/.ba?hrc      #  Idem.
~/.bashr*      #  Idem.

#  Impostando l'opzione "dotglob" si abilita anche l'espansione del punto.

#  Grazie, S.C.
```

Chapter 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.

Note: 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.

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

```
#!/bin/bash
# subshell.sh

echo

variabile_esterna=Esterna

(
variabile_interna=Interna
echo "Nella subshell, \"variabile_interna\" = $variabile_interna"
echo "Nella subshell, \"variabile_esterna\" = $variabile_esterna"
)

echo

if [ -z "$variabile_interna" ]
then
  echo "variabile_interna non definita nel corpo principale della shell"
else
  echo "variabile_interna definita nel corpo principale della shell"
fi
```

```
echo "Nel corpo principale della shell,\
\"variabile_interna\" = $variabile_interna"
# $variabile_interna viene indicata come non inizializzata perché
# le variabili definite in una subshell sono "variabili locali".

echo

exit 0
```

Vedi anche Example 32-1.

+

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

Example 20-2. Elenco dei profili utente

```
#!/bin/bash
# allprofs.sh: visualizza i profili di tutti gli utenti

# Script di Heiner Steven modificato dall'autore di questo documento.

FILE=.bashrc  #  Il file contenente il profilo utente
              #+ nello script originale era ".profile".

for home in 'awk -F: '{print $6}' /etc/passwd'
do
  [ -d "$home" ] || continue    #  Se non vi è la directory home,
                                #+ va al successivo.
  [ -r "$home" ] || continue    #  Se non ha i permessi di lettura, va
                                #+ al successivo.

  (cd $home; [ -e $FILE ] && less $FILE)
done

#  Quando lo script termina, non è necessario un 'cd' alla directory
#+ originaria, perché 'cd $home' è stato eseguito in una subshell.

exit 0
```

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

```
COMANDO1
COMANDO2
COMANDO3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMANDO4
  COMANDO5
```

```
  exit 3 # Esce solo dalla subshell.
)
# La shell genitore non è stata toccata ed il suo ambiente è preservato.
COMANDO6
COMANDO7
```

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

```
if (set -u; : $variabile) 2> /dev/null
then
  echo "La variabile è impostata."
fi     #  La variabile potrebbe essere stata impostata nello script stesso,
       #+ oppure essere una variabile interna di Bash,
       #+ oppure trattarsi di una variabile d'ambiente (che è stata esportata).

#  Si sarebbe anche potuto scrivere
#                   [[ ${variabile-x} != x || ${variabile-y} !=y ]]
# oppure            [[ ${variabile-x} != x$variabile ]]
# oppure            [[ ${variabile+x} = x ]])
```

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

```
if (set -C; : > file_lock) 2> /dev/null
then
  echo "C'è già un altro utente che sta eseguendo quello script."
  exit 65
fi

# Grazie, S.C.
```

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

Example 20-3. Eseguire processi paralleli nelle subshell

```
        (cat lista1 lista2 lista3 | sort | uniq > lista123) &
        (cat lista4 lista5 lista6 | sort | uniq > lista456) &
        # Unisce ed ordina entrambe le serie di liste simultaneamente.
        # L'esecuzione in background assicura l'esecuzione parallela.
        #
        # Stesso effetto di
        #   cat lista1 lista2 lista3 | sort | uniq > lista123 &
        #   cat lista4 lista5 lista6 | sort | uniq > lista456 &

        wait   #  Il comando successivo non viene eseguito finché le subshell
               #+ non sono terminate.

        diff lista123 lista456
```

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

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

{ comando1; comando2; comando3; ... }

Chapter 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.

Example 21-1. Eseguire uno script in modalità ristretta

```
#!/bin/bash
# Far iniziare lo script con "#!/bin/bash -r"
# significa eseguire l’intero script in modalità ristretta.

echo

echo "Cambio di directory."
cd /usr/local
echo "Ora ti trovi in ‘pwd‘"
echo "Ritorno alla directory home."
cd
echo "Ora ti trovi in ‘pwd‘"
echo

# Quello fatto fin qui è normale, modalità non ristretta.

set -r
# set --restricted    ha lo stesso significato.
echo "==> Ora lo script è in modalità ristretta. <=="
```

```
echo
echo

echo "Tentativo di cambiamento di directory in modalità ristretta."
cd ..
echo "Ti trovi ancora in ‘pwd‘"

echo
echo

echo "\$SHELL = $SHELL"
echo "Tentativo di cambiare la shell in modalità ristretta."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "Tentativo di redirigere l’output in modalità ristretta."
ls -l /usr/bin > bin.file
ls -l bin.file    #  Cerca di elencare il contenuto del file che si
                  #+ è tentato di creare.

echo

exit 0
```

Chapter 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]

> **Note:** *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.)

```
cat <(ls -l) # Uguale a     ls -l | cat

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
# Elenca tutti i file delle 3 directory principali ’bin’ e li ordina.
# Notate che a ’sort’ vengono inviati tre (contateli) distinti comandi.


diff <(comando1) <(comando2)  # Fornisce come output le differenze dei comandi.


tar cf >(bzip2 -c > file.tar.bz2) $nome_directory
# Richiama "tar cf /dev/fd/?? $nome_directory" e "bzip2 -c > file.tar.bz2".
#
# A causa della funzionalità di sistema di /dev/fd/<n>,
# non occorre che la pipe tra i due comandi sia una named pipe.
#
```

```
# Questa può essere emulata.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $nome_directory
rm pipe
#        oppure
exec 3>&1
tar cf /dev/fd/4 $nome_directory 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-


# Grazie, S.C.
```

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

```
# Frammento di script preso dalla distribuzione SuSE:

while read  des what mask iface; do
# Alcuni comandi ...
done < <(route -n)


# Per verificarlo, facciamogli fare qualcosa.
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# Output:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo


# Come ha puntualizzato S.C., una forma analoga, più facile da comprendere, è:
route -n |
  while read des what mask iface; do  # Le variabili vengono impostate
                                      #+ con l'output della pipe.
    echo $des $what $mask $iface
  done  #  Produce lo stesso output del precedente.
        #  Tuttavia, come rileva Ulrich Gayer . . .
        #+ questa forma semplificata usa una subshell per il ciclo while
        #+ e, quindi, le variabili scompaiono quando la pipe termina.
```

Notes

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

Chapter 23. Funzioni

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 considerare 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.

Example 23-1. Una semplice funzione

```
#!/bin/bash

strana ()
{
  echo "Questa è una funzione strana."
  echo "Ora usciamo dalla funzione strana."
} # La dichiarazione della funzione deve precedere la sua chiamata.

  # Ora, richiamiamo la funzione.

strana

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.

```
f1
# Dà un messaggio d’errore poiché la funzione "f1" non è stata ancora definita.

declare -f f1      # Neanche questo aiuta.
f1                 # Ancora un messaggio d’errore.

# Tuttavia...


f1 ()
{
  echo "Chiamata della funzione \"f2\" dalla funzione \"f1\"."
  f2
}

f2 ()
{
  echo "Funzione \"f2\"."
}

f1  #  La funzione "f2", in realtà, viene chiamata solo a questo punto,
    #+ sebbene vi si faccia riferimento prima della sua definizione.
    #  Questo è consentito.

    # Grazie, S.C.
```

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

```
f1 ()
{

  f2 () # annidata
  {
      echo "Funzione \"f2\", all’interno di \"f1\"."
  }

}

f2  # Restituisce un messaggio d’errore.
    # Sarebbe inutile anche farla precedere da "declare -f f2".

echo

f1  #  Non fa niente, perché richiamare"f1" non implica richiamare
    #+ automaticamente "f2".
f2  #  Ora è tutto a posto, "f2" viene eseguita, perché la sua
    #+ definizione è stata resa visibile tramite la chiamata di "f1".

    # Grazie, S.C.
```

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

```
ls -l | foo() { echo "foo"; }  # Consentito, ma inutile.


if [ "$USER" = bozo ]
then
  saluti_bozo ()   # Definizione di funzione inserita in un costrutto if/then.
  {
      echo "Ciao, Bozo."
  }
fi

saluti_bozo        # Funziona solo per Bozo, agli altri utenti dà un errore.


# Qualcosa di simile potrebbe essere utile in certi contesti.
NO_EXIT=1          # Abilita la definizione di funzione seguente.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }     #  Definizione di funzione
                                             #+ in una "lista and".
# Se $NO_EXIT è uguale a 1, viene dichiarata "exit ()".
# Così si disabilita il builtin "exit" rendendolo un alias di "true".

exit  # Viene invocata la funzione "exit ()", non il builtin "exit".

# 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.

```
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.

Example 23-2. Funzione con parametri

```
#!/bin/bash
# Funzioni e parametri

DEFAULT=predefinito               # Valore predefinito del parametro

funz2 () {
   if [ -z "$1" ]                 # Il parametro nr.1 è vuoto (lunghezza zero)?
   then
```

```
        echo "-Il parametro nr.1 ha lunghezza zero.-"  #  O non è stato passato
                                                       #+ alcun parametro.
   else
        echo "-Il parametro nr.1 è \"$1\".-"
   fi

   variabile=${1-$DEFAULT}        #  Cosa rappresenta
   echo "variabile = $variabile"  #+ la sostituzione di parametro?
                                  #  -----------------------------------
                                  #  Fa distinzione tra nessun parametro e
                                  #+ parametro nullo.

   if [ "$2" ]
   then
        echo "-Il parametro nr.2 è \"$2\".-"
   fi

   return 0
}

echo

echo "Non viene passato niente."
funz2                          # Richiamata senza alcun parametro
echo


echo "Viene passato un parametro vuoto."
funz2 ""                       # Richiamata con un parametro di lunghezza zero
echo

echo "Viene passato un parametro nullo."
funz2 "$param_non_inizializ"   # Richiamata con un parametro non inizializzato
echo

echo "Viene passato un parametro."
funz2 primo           # Richiamata con un parametro
echo

echo "Vengono passati due parametri."
funz2 primo secondo   # Richiamata con due parametri
echo

echo "Vengono passati \"\" \"secondo\"."
funz2 "" secondo      # Richiamata con il primo parametro di lunghezza zero
echo                  # e una stringa ASCII come secondo.

exit 0
```

Important: Il comando shift opera sugli argomenti passati alle funzioni (vedi Example 34-11).

Note: Rispetto ad alcuni altri linguaggi di programmazione, gli script di shell normalmente passano i parametri alle funzioni solo per valore. [1] I nomi delle variabili (che in realtà sono dei puntatori), se passati come parametri alle funzioni, vengono trattati come stringhe letterali e non possono essere dereferenziati. *Le funzioni interpretano i loro argomenti letteralmente*.

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** [2] 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 $?.

Example 23-3. Il maggiore di due numeri

```
#!/bin/bash
# max.sh: Maggiore di due numeri.

E_ERR_PARAM=-198    # Se vengono passati meno di 2 parametri alla funzione.
UGUALI=-199         # Valore di ritorno se i due numeri sono uguali.

max2 ()             # Restituisce il maggiore di due numeri.
{                   # Nota: i numeri confrontati devono essere minori di 257.
if [ -z "$2" ]
then
  return $E_ERR_PARAM
fi

if [ "$1" -eq "$2" ]
then
  return $UGUALI
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}
```

```
max2 33 34
val_ritorno=$?

if [ "$val_ritorno" -eq $E_ERR_PARAM ]
then
  echo "Bisogna passare due parametri alla funzione."
elif [ "$val_ritorno" -eq $UGUALI ]
  then
    echo "I due numeri sono uguali."
else
    echo "Il maggiore dei due numeri è $val_ritorno."
fi


exit 0

#  Esercizio (facile):
#  -------------------
#  Trasformatelo in uno script interattivo,
#+ cioè, deve essere lo script a richiedere l'input (i due numeri).
```

> **Tip:** Per fare in modo che una funzione possa restituire una stringa o un array , si deve fare ricorso ad una variabile dedicata.
>
> ```
> conteggio_righe_di_etc_passwd()
> {
>   [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
>   #  Se /etc/passwd ha i permessi di lettura, imposta REPLY al
>   #+ numero delle righe.
>   #  Restituisce o il valore del parametro o un'informazione di stato.
> }
>
> if conteggio_righe_di_etc_passwd
> then
>   echo "Ci sono $REPLY righe in /etc/passwd."
> else
>   echo "Non posso contare le righe in /etc/passwd."
> fi
>
> # Grazie, S.C.
> ```

Example 23-4. Convertire i numeri arabi in numeri Romani

```
#!/bin/bash

# Conversione di numeri arabi in numeri romani
# Intervallo: 0 - 200
# È rudimentale, ma funziona.

#  Viene lasciato come esercizio l'estensione dell'intervallo e
```

```
#+ altri miglioramenti dello script.

# Utilizzo: numero da convertire in numero romano

LIMITE=200
E_ERR_ARG=65
E_FUORI_INTERVALLO=66

if [ -z "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ numero-da-convertire"
  exit $E_ERR_ARG
fi

num=$1
if [ "$num" -gt $LIMITE ]
then
  echo "Fuori intervallo!"
  exit $E_FUORI_INTERVALLO
fi

calcola_romano ()   # Si deve dichiarare la funzione prima di richiamarla.
{
numero=$1
fattore=$2
rchar=$3
let "resto = numero - fattore"
while [ "$resto" -ge 0 ]
do
  echo -n $rchar
  let "numero -= fattore"
  let "resto = numero - fattore"
done

return $numero
       # Esercizio:
       # ----------
       # Spiegate come opera la funzione.
       # Suggerimento: divisione per mezzo di successive sottrazioni.
}


calcola_romano $num 100 C
num=$?
calcola_romano $num 90 LXXXX
num=$?
calcola_romano $num 50 L
num=$?
calcola_romano $num 40 XL
num=$?
calcola_romano $num 10 X
num=$?
calcola_romano $num 9 IX
```

```
num=$?
calcola_romano $num 5 V
num=$?
calcola_romano $num 4 IV
num=$?
calcola_romano $num 1 I

echo

exit 0
```

Vedi anche Example 10-28.

Important: Il più grande intero positivo che una funzione può restituire è 256. 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 256.

Example 23-5. Verificare valori di ritorno di grandi dimensioni in una funzione

```
#!/bin/bash
# return-test.sh

# Il maggiore valore positivo che una funzione può restituire è 255.

val_ritorno ()         # Restituisce tutto quello che gli viene passato.
{
  return $1
}

val_ritorno 27         # o.k.
echo $?                # Restituisce 27.

val_ritorno 255        # Ancora o.k.
echo $?                # Restituisce 255.

val_ritorno 257        # Errore!
echo $?                # Restituisce 1 (codice d'errore generico).

# ============================================================
val_ritorno -151896    # Funziona con grandi numeri negativi?
echo $?                # Restituirà -151896?
                       # No! Viene restituito 168.
#  Le versioni di Bash precedenti alla 2.05b permettevano
#+ valori di ritorno di grandi numeri negativi.
#  Quelle più recenti non consentono questa scappatoia.
#  Ciò potrebbe rendere malfunzionanti i vecchi script.
#  Attenzione!
# ============================================================

exit 0
```

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

```
Val_Ritorno=  #  Variabile globale che contiene un valore di ritorno
```

```
             #+ della funzione maggiore di 256.

ver_alt_ritorno ()
{
  fvar=$1
  Val_Ritorno=$fvar
  return      # Restituisce 0 (successo).
}

ver_alt_ritorno 1
echo $?                                     # 0
echo "valore di ritorno = $Val_Ritorno"    # 1

ver_alt_ritorno 256
echo "valore di ritorno = $Val_Ritorno"    # 256

ver_alt_ritorno 257
echo "valore di ritorno = $Val_Ritorno"    # 257

ver_alt_ritorno 25701
echo "valore di ritorno = $Val_Ritorno"    # 25701
```

Un metodo perfino più elegante consiste semplicemente nell'usare il "valore di ritorno" della funzione con il comando **echo** e "catturarlo" successivamente per mezzo della sostituzione di parametro. Per una discussione sull'argomento vedi Section 34.7.

Example 23-6. Confronto di due interi di grandi dimensioni

```
#!/bin/bash
# max2.sh: Maggiore di due GRANDI interi.

#  È il precedente esempio "max.sh" ,
#+ modificato per consentire il confronto di grandi numeri.

UGUALI=0            # Valore di ritorno se i due parametri sono uguali.
E_ERR_PARAM=-99999  # Numero di parametri passati alla funzione insufficiente.

max2 ()             # "Restituisce" il maggiore di due numeri.
{
if [ -z "$2" ]
then
  echo $E_ERR_PARAM
  return
fi

if [ "$1" -eq "$2" ]
then
  echo $UGUALI
  return
else
  if [ "$1" -gt "$2" ]
  then
      valritorno=$1
  else
      valritorno=$2
  fi
fi
```

```
echo $valritorno    # Visualizza il valore invece di restituirlo.

}


val_ritorno=$(max2 33001 33997)
# Sostituzione di parametro avanzata.
# Assegna lo stdout della funzione alla variabile 'val_ritorno' . . .


if [ "$val_ritorno" -eq "$E_ERR_PARAM" ]
then
  echo "Errore nel numero di parametri passati alla funzione di confronto!"
elif [ "$val_ritorno" -eq "$UGUALI" ]
  then
      echo "I due numeri sono uguali."
  else
      echo "Il maggiore dei due numeri è $val_ritorno."
fi

exit 0

#  Esercizio:
#  ---------
#  Trovate un modo più elegante per verificare
#+ il numero di parametri passati alla funzione.
```

Vedi anche Example 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 256.

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 Example 3-1).

Example 23-7. Il vero nome dal nome utente

```
#!/bin/bash

# Partendo dal nome dell'utente, ricava il "vero nome" da /etc/passwd.

CONTOARG=1  # Si aspetta un argomento.
E_ERR_ARG=65

file=/etc/passwd
modello=$1

if [ $# -ne "$CONTOARG" ]
```

```
then
  echo "Utilizzo: ‘basename $0‘ NOME-UTENTE"
  exit $E_ERR_ARG
fi

ricerca ()      # Esamina il file alla ricerca del modello, una riga per volta.
{
while read riga # while non necessariamente vuole la "[ condizione]"
do
  echo "$riga" | grep $1 | awk -F":" ’{ print $5 }’ #  awk deve usare
                                                     #+ i ":" come delimitatore.
done
} <$file  # Redirige nello stdin della funzione.

ricerca $modello

# Certo, l’intero script si sarebbe potuto ridurre a
#       grep MODELLO /etc/passwd | awk -F":" ’{ print $5 }’
# oppure
#       awk -F: ’/MODELLO/ {print $5}’
# oppure
#       awk -F: ’($1 == "nomeutente") { print $5 }’ #  il vero nome dal
                                                     #+ nome utente
# Tuttavia, non sarebbe stato altrettanto istruttivo.

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.

```
# Invece di:
Funzione ()
{
 ...
} < file

# Provate:
Funzione ()
{
  {
    ...
   } < file
}

# Analogamente,

Funzione ()  # Questa funziona.
{
  {
  echo $*
  } | tr a b
}
```

```
Funzione ()  # Questa, invece, no.
{
  echo $*
} | tr a b   # In questo caso è obbligatorio il blocco di codice annidato.


# Grazie, S.C.
```

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.

Example 23-8. Visibilità di una variabile locale

```
#!/bin/bash
# Variabili globali e locali in una funzione.

funz ()
{
  local var_locale=23       # Dichiarata come variabile locale.
  echo                      # Utilizza il builtin ’local’.
  echo "\"var_locale\" nella funzione = $var_locale"
  var_globale=999           # Non dichiarata come locale.
                            # Viene impostata per default a globale.
  echo "\"var_globale\" nella funzione = $var_globale"
}

funz

#  Ora, per controllare se la variabile locale "var_locale" esiste al di fuori
#+ della funzione.

echo
echo "\"var_locale\" al di fuori della funzione = $var_locale"
                              # $var_locale al di fuori della funzione =
                              # No, $var_locale non ha visibilità globale.
echo "\"var_globale\" al di fuori della funzione = $var_globale"
                              # $var_globale al di fuori della funzione = 999
                              # $var_globale è visibile globalmente
echo
```

```
exit 0
#  A differenza del C, una variabile Bash dichiarata all'interno di una funzione
#+ è locale "solo" se viene dichiarata come tale.
```

Caution

Prima che una funzione venga chiamata, *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*.

```
#!/bin/bash

funz ()
{
var_globale=37    #  Visibile solo all'interno del blocco della funzione
                  #+ prima che la stessa venga richiamata.
}                 #  FINE DELLA FUNZIONE

echo "var_globale = $var_globale"  #  var_globale =
                                   #  La funzione "funz" non è ancora stata
                                   #+ chiamata, quindi $var_globale qui non è
                                   #+ visibile.

funz
echo "var_globale = $var_globale"  # var_globale = 37
                                   # È stata impostata richiamando la funzione.
```

23.2.1. Le variabili locali aiutano a realizzare la ricorsività.

Le variabili locali consentono la ricorsività, [3] ma questa pratica implica, generalmente, un carico computazionale elevato e, in definitiva, *non* viene raccomandata in uno script di shell. [4]

Example 23-9. Ricorsività tramite una variabile locale

```
#!/bin/bash

#               fattoriale
#               ----------


# Bash permette la ricorsività?
# Ebbene, sì, ma...
# Dovreste avere dei sassi al posto del cervello per usarla.


MAX_ARG=5
E_ERR_ARG=65
E_ERR_MAXARG=66


if [ -z "$1" ]
```

```
then
  echo "Utilizzo: ‘basename $0‘ numero"
  exit $E_ERR_ARG
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Valore troppo grande (il massimo è 5)."
  #  Torniamo alla realtà.
  #  Se fosse necessario un numero maggiore di questo,
  #+ riscrivete lo script in un vero linguaggio di programmazione.
  exit $E_ERR_MAXARG
fi

fatt ()
{
  local numero=$1
  #  La variabile "numero" deve essere dichiarata locale,
  #+ altrimenti questa funzione non svolge il suo compito.
  if [ "$numero" -eq 0 ]
  then
      fattoriale=1    # Il fattoriale di 0 è 1.
  else
      let "decrnum = numero - 1"
      fatt $decrnum  # Chiamata ricorsiva della funzione.
      let "fattoriale = $numero * $?"
   fi

   return $fattoriale
}

fatt $1
echo "Il fattoriale di $1 è $?."

exit 0
```

Vedi anche Example 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.

23.3. Ricorsività senza variabili locali

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

Example 23-10. La torre di Hanoi

```
#! /bin/bash
#
# La Torre di Hanoi
# Script Bash
```

```
# Copyright (C) 2000 Amit Singh. Tutti i diritti riservati.
# http://hanoi.kernelthread.com
#
# Ultima verifica eseguita con la versione bash 2.05b.0(13)-release
#
#  Usato in "Advanced Bash Scripting Guide"
#+ con il permesso dell'autore dello script.
#  Commentato e leggermente modificato dall'autore di ABS.

#==========================================================================#
#  La Torre di Hanoi è un vecchio rompicapo matematico.
#  Ci sono tre pioli verticali inseriti in una base.
#  Nel primo piolo è impilata una serie di anelli rotondi.
#  Gli anelli sono dei dischi piatti con un foro al centro,
#+ in modo che possano essere infilati nei pioli.
#  I dischi hanno diametri diversi e sono impilati in ordine
#+ decrescente in base alla loro dimensione.
#  Quello più piccolo si trova nella posizione più alta,
#+ quello più grande alla base.
#
#  Lo scopo è quello di trasferire la pila di dischi
#+ in uno degli altri pioli.
#  Si può spostare solo un disco alla volta.
#  È consentito rimettere i dischi nel piolo iniziale.
#  È permesso mettere un disco su un altro di dimensione maggiore,
#+ ma *non* viceversa.
#  Ancora, è proibito collocare un disco su uno di minor diametro.
#
#  Con un numero ridotto di dischi, sono necessari solo pochi spostamenti.
#+ Per ogni disco aggiuntivo,
#+ il numero degli spostamenti richiesti approssimativamente raddoppia
#+ e la "strategia" diventa sempre più complessa.
#
#  Per ulteriori informazioni, vedi http://hanoi.kernelthread.com.
#
#
#         ...                    ...                    ...
#         | |                    | |                    | |
#        _|_|_                   | |                    | |
#       |_____|                  | |                    | |
#      |_______|                 | |                    | |
#     |_________|                | |                    | |
#    |___________|               | |                    | |
#   |             |              | |                    | |
# .-------------------------------------------------------------.
# |*************************************************************|
#          #1                    #2                     #3
#
#==========================================================================#


E_NOPARAM=66    # Nessun parametro passato allo script.
E_ERR_PARAM=67  # Il numero di dischi passato allo script non è valido.
```

```
Mosse=          # Variabile globale contenente il numero degli spostamenti.
                # Modifiche allo script originale.

eseguehanoi() { # Funzione ricorsiva.
    case $1 in
    0)
        ;;
    *)
        eseguehanoi "$(($1-1))" $2 $4 $3
        echo sposto $2 "-->" $3
let "Mosse += 1"  # Modifica allo script originale.
        eseguehanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

case $# in
1)
    case $(($1>0)) in     # Deve esserci almeno un disco.
    1)
        eseguehanoi $1 1 3 2
        echo "Totale spostamenti = $Mosse"
        exit 0;
        ;;
    *)
        echo "$0: numero di dischi non consentito";
        exit $E_ERR_PARAM;
        ;;
    esac
    ;;
*)
    echo "utilizzo: $0 N"
    echo "       Dove \"N\" è il numero dei dischi."
    exit $E_NOPARAM;
    ;;
esac

# Esercizi:
# ---------
# 1) Eventuali comandi posti in questo punto verrebbero eseguiti?
#    Perché no? (Facile)
# 2) Spiegate il funzionamento della funzione "eseguehanoi".
#    (Difficile)
```

Notes

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

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

```
ITERAZIONI=3  # Numero di input da immettere.
contai=1

lettura () {
  # Richiamata nella forma lettura nomevariabile,
  # visualizza il dato precedente tra parentesi quadre come dato predefinito,
  # quindi chiede un nuovo valore.

  local var_locale

  echo -n "Inserisci un dato "
  eval 'echo -n "[$'$1'] "'  # Dato precedente.
  read var_locale
  [ -n "$var_locale" ] && eval $1=\$var_locale

  #  "Lista And": se "var_locale" è presente allora viene impostata
  #+ al valore di "$1".
}

echo

while [ "$contai" -le "$ITERAZIONI" ]
do
  lettura var
  echo "Inserimento nr.$contai = $var"
  let "contai += 1"
  echo
done


# Grazie a Stephane Chazelas per aver fornito questo istruttivo esempio.

exit 0
```

2. Il comando **return** è un builtin Bash.
3. 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.
4. Troppi livelli di ricorsività possono mandare in crash lo script con un messaggio di segmentation fault.

```
#!/bin/bash

#  Attenzione: è probabile che l'esecuzione di questo script blocchi il sistema!
#  Se siete fortunati, verrete avvertiti da un segmentation fault prima che
#+ tutta la memoria disponibile venga occupata.

funzione_ricorsiva ()
{
(( $1 < $2 )) && f $(( $1 + 1 )) $2
#  Finché il primo parametro è inferiore al secondo,
#+ il primo viene incrementato ed il tutto si ripete.
```

```
}

funzione_ricorsiva 1 50000  # Ricorsività di 50,000 livelli!
#  Molto probabilmente segmentation fault (in base alla dimensione dello stack,
#+ impostato con ulimit -m).

#  Una ricorsività così elevata potrebbe causare un segmentation fault
#+ anche in un programma in C, a seguito dell'uso di tutta la memoria
#+ allocata nello stack.


echo "Probabilmente questo messaggio non verrà visualizzato."
exit 0  # Questo script non terminarà normalmente.

#  Grazie, Stephane Chazelas.
```

Chapter 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**. Ciò può risparmiare una grande quantità di digitazione 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.

Example 24-1. Alias in uno script

```
#!/bin/bash
#  Invocatelo con un parametro da riga di comando per provare l'ultima sezione
#+ dello script.

shopt -s expand_aliases
#  È necessario impostare questa opzione, altrimenti lo script non espande
#+ gli alias.


# Innazitutto, divertiamoci un po'.
alias Jesse_James='echo "\"Alias Jesse James\" era una commedia del 1959\
interpretata da Bob Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
#  Per definire un alias si possono usare sia gli apici singoli (') che quelli
#+ doppi (").

echo "Prova dell'alias \"ll\":"
ll /usr/X11R6/bin/mk*   #* L'alias funziona.

echo

directory=/usr/X11R6/bin/
prefisso=mk*  # Vediamo se il carattere jolly causa dei problemi.
echo "Variabili \"directory\" + \"prefisso\" = $directory$prefisso"
echo

alias lll="ls -l $directory$prefisso"

echo "Prova dell'alias \"lll\":"
```

```
lll           #  Lungo elenco di tutti i file presenti in /usr/X11R6/bin che
              #+ iniziano con mk.
# L’alias gestisce le variabili concatenate e il carattere jolly, o.k.



TRUE=1

echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "Prova dell’alias \"rr\" all’interno di un enunciato if/then:"
  rr /usr/X11R6/bin/mk*   #* Messaggio d’errore!
  # Gli alias non vengono espansi all’interno di enunciati composti.
  echo "Comunque, l’alias precedentemente espanso viene ancora riconosciuto:"
  ll /usr/X11R6/bin/mk*
fi

echo

conto=0
while [ $conto -lt 3 ]
do
  alias rrr="ls -l"
  echo "Prova dell’alias \"rrr\" in un ciclo \"while\":"
  rrr /usr/X11R6/bin/mk*  #* Anche in questo caso l’alias non viene espanso.
                          #  alias.sh: line 57: rrr: command not found
  let conto+=1
done

echo; echo

alias xyz=’cat $0’   # Lo script visualizza sé stesso.
                     # Notate il quoting forte.
xyz
#  Questo sembra funzionare,
#+ sebbene la documentazione Bash suggerisca il contrario.
#
#  In ogni caso, come ha evidenziato Steve Jacobson,
#+ il parametro "$0" viene espanso immediatamente alla
#+ dichiarazione dell’alias.

exit 0
```

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

Example 24-2. unalias: Abilitare e disabilitare un alias

```
#!/bin/bash

shopt -s expand_aliases  # Abilita l'espansione dell'alias.

alias llm='ls -al | more'
llm

echo

unalias llm              # Disabilita l'alias.
llm
# Dà un messaggio d'errore poiché 'llm' non viene più riconosciuto.

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
```

Notes

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

Chapter 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

```
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).

Example 25-1. Usare una “lista and” per verificare gli argomenti da riga di comando

```
#!/bin/bash
# "lista and"

if [ ! -z "$1" ] && echo "Argomento nr.1 = $1" && [ ! -z "$2" ] &&\
echo "Argomento nr.2 = $2"

then
  echo "Allo script sono stati passati almeno 2 argomenti."
  # Tutti i comandi della serie hanno restituito true.
else
  echo "Allo script sono stati passati meno di 2 argomenti."
  # Almeno uno dei comandi ha restituito false.
fi
# Notate che "if [ ! -z $1 ]" funziona, ma il suo supposto equivalente,
#  if [ -n $1 ] no. Comunque, l’uso del quoting risolve il problema.
#  if [ -n "$1" ] funziona.  State attenti!
# In una verifica, è sempre meglio usare le variabili con il quoting.


# Questo svolge la stesso compito usando solamente enunciati if/then.
if [ ! -z "$1" ]
then
  echo "Argomento nr.1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Argomento nr.2 = $2"
  echo "Allo script sono stati passati almeno 2 argomenti."
else
  echo "Allo script sono stati passati meno di 2 argomenti."
fi
# È più lungo e meno elegante di una "lista and".
```

```
exit 0
```

Example 25-2. Un’altra verifica di argomenti da riga di comando utilizzando una “lista and”

```
#!/bin/bash

ARG=1         # Numero degli argomenti attesi.
E_ERR_ARG=65  # Valore d’uscita se il numero di argomenti passati è errato.

test $# -ne $ARG && echo "Utilizzo: ‘basename $0‘ $ARG \
argomento/i" && exit $E_ERR_ARG
#  Se la prima condizione è vera (numero errato di argomenti passati allo
#+ script), allora vengono eseguiti i comandi successivi e lo script termina.

# La riga seguente verrà eseguita solo se fallisce la verifica precedente.
echo "Allo script è stato passato un numero corretto di argomenti."

exit 0

#  Per verificare il valore d’uscita, eseguite "echo $?" dopo che lo script
#+ è terminato.
```

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

```
arg1=$@       #  Imposta $arg1 al numero di argomenti passati da riga di
              #+ comando, se ce ne sono.

[ -z "$arg1" ] && arg1=DEFAULT
              #  Viene impostata a DEFAULT se, da riga di comando, non è
              #+ stato passato niente.
```

lista or

```
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”.

Example 25-3. Utilizzare la “lista or” in combinazione con una “lista and”

```
#!/bin/bash

#  delete.sh, utility di cancellazione di file non molto intelligente.
#  Utilizzo: delete nomefile

E_ERR_ARG=65
```

```
if [ -z "$1" ]
then
  echo "Utilizzo: ‘basename $0‘ nomefile"
  exit $E_ERR_ARG  # Nessun argomento? Abbandono.
else
  file=$1          # Imposta il nome del file.
fi


[ ! -f "$file" ] && echo "File \"$file\" non trovato. \
Mi rifiuto, in modo vile, di cancellare un file inesistente."
#  LISTA AND, fornisce il messaggio d’errore se il file non è presente.
#  Notate il messaggio di echo che continua alla riga successiva per mezzo del
#+ carattere di escape.

[ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" cancellato.")
# LISTA OR, per cancellare il file se presente.

# Notate l’inversione logica precedente.
# La LISTA AND viene eseguita se il risultato è true, la LISTA OR se è false.

exit 0
```

> **Caution**
>
> Se il primo comando di una “lista or” restituisce true, esso verrà *comunque* eseguito.

Important: 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.

```
false && true || echo false         # false

# Stesso risultato di
( false && true ) || echo false     # false
# Ma *non*
false && ( true || echo false )     # (non viene visualizzato niente)

#  Notate i raggruppamenti e la valutazione degli enunciati da sinistra a destra
#+ perché gli operatori logici "&&" e "||" hanno la stessa priorità.

#  È meglio evitare tali complessità, a meno che non sappiate cosa state facendo

#  Grazie, S.C.
```

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

Chapter 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]}`**.

Example 26-1. Un semplice uso di array

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFO

# Non occorre che gli elementi dell'array siano consecutivi o contigui.

# Alcuni elementi possono rimanere non inizializzati
# I "buchi" negli array sono permessi


echo -n "area[11] = "
echo ${area[11]}    #  sono necessarie le {parentesi graffe}

echo -n "area[13] = "
echo ${area[13]}

echo "Il contenuto di area[51] è ${area[51]}."

# Gli elementi non inizializzati vengono visualizzati come spazi.
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43] non assegnato)"

echo

# Somma di due elementi dell'array assegnata ad un terzo
area[5]=‘expr ${area[11]} + ${area[13]}‘
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=‘expr ${area[11]} + ${area[51]}‘
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
#  Questo assegnamento fallisce perché non è permesso sommare
#+ un intero con una stringa.

echo; echo; echo
```

```
# --------------------------------------------------------------------------
# Un altro array, "area2".
# Metodo di assegnamento alternativo...
# nome_array=( XXX YYY ZZZ ... )

area2=( zero uno due tre quattro )

echo -n "area2[0] = "
echo ${area2[0]}
#  Aha, indicizzazione in base zero (il primo elemento dell'array
#+ è [0], non [1]).

echo -n "area2[1] = "
echo ${area2[1]}    # [1] è il secondo elemento dell'array.
# --------------------------------------------------------------------------

echo; echo; echo

# ---------------------------------------------
# Ancora un altro array, "area3".
# Ed un'altra modalità ancora di assegnamento...
# nome_array=([xx]=XXX [yy]=YYY ...)

area3=([17]=diciassette [24]=ventiquattro)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[24] = "
echo ${area3[24]}
# ---------------------------------------------

exit 0
```

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

```
stringa=abcABC123ABCabc
echo ${stringa[@]}              # abcABC123ABCabc
echo ${stringa[*]}              # abcABC123ABCabc
echo ${stringa[0]}              # abcABC123ABCabc
echo ${stringa[1]}              # Nessun output!
                                # Perchè?
echo ${#stringa[@]}             # 1
                                # Array di un solo elemento.
                                # La stringa stessa.

# Grazie a Michael Zick per la precisazione.
```

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

Example 26-2. Impaginare una poesia

```
#!/bin/bash
# poem.sh: Visualizza in modo elegante una delle poesie preferite dall'autore.

# Righe della poesia (una strofa).
Riga[1]="I do not know which to prefer,"
Riga[2]="The beauty of inflections"
Riga[3]="Or the beauty of innuendoes,"
Riga[4]="The blackbird whistling"
Riga[5]="Or just after."

# Attribuzione.
Attrib[1]=" Wallace Stevens"
Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
# La poesia è di Dominio Pubblico (copyright scaduto).

for indice in 1 2 3 4 5    # Cinque righe.
do
  printf "     %s\n" "${Riga[indice]}"
done

for indice in 1 2          # Attribuzione di due righe.
do
  printf "          %s\n" "${Attrib[indice]}"
done

exit 0
```

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.

Example 26-3. Operazioni diverse sugli array

```
#!/bin/bash
# array-ops.sh: Un po' di divertimento con gli array.


array=( zero uno due tre quattro cinque )

echo ${array[0]}       #  zero
echo ${array:0}        #  zero
                       #  Espansione di parametro del primo elemento,
                       #+ iniziando dalla posizione nr. 0 (1° carattere).
echo ${array:1}        #  ero
                       #  Espansione di parametro del primo elemento,
                       #+ iniziando dalla posizione nr. 1 (2° carattere).

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

echo ${#array[0]}      #  4
                       #  Lunghezza del primo elemento dell'array.
echo ${#array}         #  4
```

```
                      #  Lunghezza del primo elemento dell'array.
                      #  (Notazione alternativa)

echo ${#array[1]}     #  3
                      #  Lunghezza del secondo elemento dell'array.
       #  Gli array in Bash sono indicizzati in base zero.

echo ${#array[*]}     #  6
                      #  Numero di elementi di array.
echo ${#array[@]}     #  6
                      #  Numero di elementi di array.

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

array2=( [0]="primo elemento" [1]="secondo elemento" [3]="quarto elemento" )

echo ${array2[0]}     # primo elemento
echo ${array2[1]}     # secondo elemento
echo ${array2[2]}     #
                      # Saltato durante l'inizializzazione, quindi nullo.
echo ${array2[3]}     # quarto elemento


exit 0
```

Con gli array funzionano anche molte delle normali operazioni stringa.

Example 26-4. Operazioni di stringa con gli array

```
#!/bin/bash
# array-strops.sh: Operazioni su stringhe negli array.
# Script di Michael Zick.
# Usato con il permesso dell'autore.

#  In generale, qualsiasi operazione stringa nella notazione ${nome ... }
#+ può essere applicata a tutti gli elementi stringa presenti in un array
#+ usando la notazione ${nome[@] ... } o ${nome[*] ...}.


arrayZ=( uno due tre quattro cinque cinque )

echo

# Estrazione di sottostringa successiva
echo ${arrayZ[@]:0}     # uno due tre quattro cinque cinque
                        # Tutti gli elementi.

echo ${arrayZ[@]:1}     # due tre quattro cinque cinque
                        # Tutti gli elementi successivi ad elemento[0].

echo ${arrayZ[@]:1:2}   # due tre
                        # Solo i due elementi successivi ad elemento[0].
```

```
echo "-----------------------"

#  Rimozione di sottostringa
#  Rimuove l'occorrenza più breve dalla parte iniziale della(e) stringa(he),
#+ dove sottostringa è un'espressione regolare.

echo ${arrayZ[@]#q*o}   # uno due tre cinque cinque
                        # Controlla tutti gli elementi dell'array.
                        # Verifica "quattro" e lo rimuove.

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

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

# 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 una chiamata di funzione.
#   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.

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

```
#!/bin/bash
# script-array.sh: Inserisce questo stesso script in un array.
# Ispirato da una e-mail di Chris Martin (grazie!).

contenuto_script=( $(cat "$0") ) #  Registra il contenuto di questo script ($0)
                                 #+ in un array.

for elemento in $(seq 0 $((${#contenuto_script[@]} - 1)))
  do                #  ${#contenuto_script[@]}
                    #+ fornisce il numero degli elementi di un array.
                    #
                    #  Domanda:
                    #  Perché è necessario  seq 0?
                    #  Provate a cambiarlo con seq 1.
  echo -n "${contenuto_script[$elemento]}"
                    # Elenca tutti i campi dello script su una sola riga.
  echo -n " -- "    # Usa " -- " come separatore di campo.
done
```

```
echo

exit 0

# Esercizio:
# ---------
#  Modificate lo script in modo che venga visualizzato
#+ nella sua forma originale,
#+ completa di spazi, interruzioni di riga, ecc.
```

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

Example 26-6. Alcune proprietà particolari degli array

```
#!/bin/bash

declare -a colori
# Consente di dichiarare un array senza doverne specificare la dimensione.

echo "Inserisci i tuoi colori preferiti (ognuno separato da uno spazio)."

read -a colori    #  Inserite almeno 3 colori per mettere alla prova le
                  #+ funzionalità che seguono.
#  Opzione speciale del comando 'read',
#+ che consente l'assegnamento degli elementi di un array.

echo

conta_elementi=${#colori[@]}
# Sintassi speciale per ricavare il numero di elementi di un array.
#     conta_elementi=${#colori[*]} anche in questo modo.
#
#  La variabile "@" permette la suddivisione delle parole, anche se all'interno
#+ degli apici (estrae le variabili separate da spazi).

indice=0

while [ "$indice" -lt "$conta_elementi" ]
do    # Elenca tutti gli elementi dell' array.;
  echo ${colori[$indice]}
  let "indice = $indice + 1"
done
# Ogni elemento dell'array viene visualizzato su una riga singola.
# Se non vi piace, utilizzate  echo -n "${colori[$indice]} "
#
# La stessa cosa utilizzando un ciclo "for":
#   for i in "${colori[@]}"
#   do
#     echo "$i"
#   done
# (Grazie, S.C.)
```

```
echo

#  Ancora, elenco di tutti gli elementi dell'array utilizzando, però, un
#+ metodo più elegante.
  echo ${colori[@]}         # anche echo ${colori[*]}.

echo

# Il comando "unset" cancella gli elementi di un array, o l'intero array.
unset colori[1]             # Cancella il secondo elemento dell' array.
                            # Stesso effetto di colori[1]=
echo  ${colori[@]}          # Elenca ancora l'array. Manca il secondo elemento.

unset colori                # Cancella l'intero array.
                            #  Anche: unset colori[*] e
                            #+ unset colori[@].
echo; echo -n "Colori cancellati."
echo ${colori[@]}           # Visualizza ancora l'array, ora vuoto.
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.

Example 26-7. Array vuoti ed elementi vuoti

```
#!/bin/bash
# empty-array.sh

#  Grazie a Stephane Chazelas, per l'esempio originario,
#+ e a Michael Zick, per averlo ampliato.


# Un array vuoto non è la stessa cosa di un array composto da elementi vuoti.

array0=( primo secondo terzo )
array1=( ” )   # "array1" ha un elemento vuoto.
array2=( )      # Nessun elemento... "array2" è vuoto.

echo
ElencaArray ()
{
echo "Elementi in array0:  ${array0[@]}"
echo "Elementi in array1:  ${array1[@]}"
echo "Elementi in array2:  ${array2[@]}"
echo
echo "Lunghezza del primo elemento di array0 = ${#array0}"
echo "Lunghezza del primo elemento di array1 = ${#array1}"
echo "Lunghezza del primo elemento di array2 = ${#array2}"
echo
echo "Numero di elementi di array0 = ${#array0[*]}"  # 3
```

```
echo "Numero di elementi di array1 = ${#array1[*]}"  # 1  (sorpresa!)
echo "Numero di elementi di array2 = ${#array2[*]}"  # 0
}

# ==================================================================

ElencaArray

# Proviamo ad incrementare gli array

# Aggiunta di un elemento ad un array.
array0=( "${array0[@]}" "nuovo1" )
array1=( "${array1[@]}" "nuovo1" )
array2=( "${array2[@]}" "nuovo1" )

ElencaArray

# oppure
array0[${#array0[*]}]="nuovo2"
array1[${#array1[*]}]="nuovo2"
array2[${#array2[*]}]="nuovo2"

ElencaArray

# Quando sono modificati in questo modo, gli array sono come degli ’stack’
# L’operazione precedente rappresenta un ’push’
# L”altezza’ dello stack è:
altezza=${#array2[@]}
echo
echo "Altezza dello stack array2 = $altezza"

# Il ’pop’ è:
unset array2[${#array2[@]}-1] # Gli array hanno indici in base zero
altezza=${#array2[@]}
echo
echo "POP"
echo "Nuova altezza dello stack array2 = $altezza"

ElencaArray

# Elenca solo gli elemnti 2do e 3zo dell’array0
da=1 # Numerazione in base zero
a=2 #
declare -a array3=( ${array0[@]:1:2} )
echo
echo "Elementi dell’array3:  ${array3[@]}"

# Funziona come una stringa (array di caratteri)
# Provate qualche altro tipo di "stringa"

# Sostituzione
declare -a array4=( ${array0[@]/secondo/2do} )
echo
```

```
echo "Elementi dell’array4:  ${array4[@]}"

# Sostituzione di ogni occorrenza della stringa con il carattere jolly
declare -a array5=( ${array0[@]//nuovo?/vecchio} )
echo
echo "Elementi dell’array5:  ${array5[@]}"

# Proprio quando stavate per prenderci la mano...
declare -a array6=( ${array0[@]#*nuovo} )
echo # Questo potrebbe sorprendervi
echo "Elementi dell’array6:  ${array6[@]}"

declare -a array7=( ${array0[@]#nuovo1} )
echo # Dopo l’array6 questo non dovrebbe più stupirvi
echo "Elementi dell’array7:  ${array7[@]}"

# Che assomiglia moltissimo a...
declare -a array8=( ${array0[@]/nuovo1/} )
echo
echo "Elementi dell’array8:  ${array8[@]}"

#  Quindi, cosa possiamo dire a questo proposito?

#  Le operazioni stringa vengono eseguite su ognuno
#+ degli elementi presenti in var[@] in sequenza.
#  Quindi : BASH supporta le operazioni su vettore stringa
#  Se il risultato è una stringa di lunghezza zero,
#+ quell’elemento scompare dall’assegnamento risultante.

#  Domanda: queste stringhe vanno usate con il quoting forte o debole?

zap=’nuovo*’
declare -a array9=( ${array0[@]/$zap/} )
echo
echo "Elementi dell’array9:  ${array9[@]}"

# Proprio quando pensavate di essere a cavallo...
declare -a array10=( ${array0[@]#$zap} )
echo
echo "Elementi dell’array10:  ${array10[@]}"

# Confrontate l’array7 con l’array10
# Confrontate l’array8 con l’array9

# Risposta: quoting debole.

exit 0
```

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

```
# Copiare un array.
array2=( "${array1[@]}" )
```

```
# oppure
array2="${array1[@]}"

# Aggiunta di un elemento ad un array.
array=( "${array[@]}" "nuovo elemento" )
# oppure
array[${#array[*]}]="nuovo elemento"

# Grazie, S.C.
```

> **Tip:** 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.
>
> ```
> #!/bin/bash
>
> nomefile=file_esempio
>
> #            cat file_esempio
> #
> #            1 a b c
> #            2 d e fg
>
>
> declare -a array1
>
> array1=( ‘cat "$nomefile" | tr ’\n’ ’ ’’)  # Carica il contenuto
>                                            # di $nomefile in array1.
> #         visualizza il file allo stdout,
> #         cambiando i ritorni a capo presenti nel file in spazi.
>
> echo ${array1[@]}           # Visualizza il contenuto dell’array.
> #                             1 a b c 2 d e fg
> #
> #  Ogni "parola" separata da spazi presente nel file
> #+ è stata assegnata ad un elemento dell’array.
>
> conta_elementi=${#array1[*]}
> echo $conta_elementi        # 8
> ```

Uno scripting intelligente consente di aggiungere ulteriori operazioni sugli array.

Example 26-8. Initializzare gli array

```
#! /bin/bash
# array-assign.bash

#  Le operazioni degli array sono specifiche di Bash,
#+ quindi il nome dello script deve avere il suffisso ".bash".
```

```
# Copyright (c) Michael S. Zick, 2003, Tutti i diritti riservati.
# Licenza: Uso illimitato in qualsiasi forma e per qualsiasi scopo.
# Versione: $ID$

#  Basato su un esempio fornito da Stephane Chazelas,
#+ apparso nel libro: Guida avanzata di bash scripting.

# Formato dell'output del comando 'times':
# CPU Utente <spazio> CPU Sistema
# CPU utente di tutti i processi <spazio> CPU sistema di tutti i processi

#  Bash possiede due modi per assegnare tutti gli elementi di un array
#+ ad un nuovo array.
#  Nelle versioni Bash 2.04, 2.05a e 2.05b.
#+ entrambi i metodi inseriscono gli elementi 'nulli'
#  Alle versioni più recenti può aggiungersi un ulteriore assegnamento
#+ purché, per tutti gli array, sia mantenuta la relazione [indice]=valore.

#  Crea un array di grandi dimensioni utilizzando un comando interno,
#+ ma andrà bene qualsiasi cosa che permetta di creare un array
#+ di diverse migliaia di elementi.

declare -a grandePrimo=( /dev/* )
echo
echo 'Condizioni: Senza quoting, IFS preimpostato, Tutti gli elementi'
echo "Il numero di elementi dell'array è ${#grandePrimo[@]}"

# set -vx



echo
echo '- - verifica: =( ${array[@]} ) - -'
times
declare -a grandeSecondo=( ${grandePrimo[@]} )
#                          ^                   ^
times

echo
echo '- - verifica: =${array[@]} - -'
times
declare -a grandeTerzo=${grandePrimo[@]}
# Questa volta niente parentesi.
times

#  Il confronto dei risultati dimostra che la seconda forma, evidenziata
#+ da Stephane Chazelas, è da tre a quattro volte più veloce.

#  Nei miei esempi esplicativi, continuerò ad utilizzare la prima forma
#+ perché penso serva ad illustrare meglio quello che avviene.

#  In realtà, porzioni di codice di miei esempi possono contenere
#+ la seconda forma quando è necessario velocizzare l'esecuzione.
```

```
# MSZ: Scusate le precedenti sviste.


exit 0
```

Example 26-9. Copiare e concatenare array

```
#! /bin/bash
# CopyArray.sh
#
# Script di Michael Zick.
# Usato con il permesso dell'autore.

#  Come "Passare per Nome & restituire per Nome"
#+ ovvero "Costruirsi il proprio enunciato di assegnamento".


CpArray_Mac() {

# Costruttore dell'enunciato di assegnamento

    echo -n 'eval '
    echo -n "$2"                    # Nome di destinazione
    echo -n '=( ${'
    echo -n "$1"                    # Nome di origine
    echo -n '[@]} )'

# Si sarebbe potuto fare con un solo comando.
# E' solo una questione di stile.
}

declare -f CopiaArray               # Funzione "Puntatore"
CopiaArray=CpArray_Mac              # Costruttore dell'ennuciato

Enfatizza()
{

# Enfatizza l'array di nome $1.
# (Lo sposa all'array contenente "veramente fantastico".)
# Risultato nell'array di nome $2.

    local -a TMP
    local -a esagerato=( veramente fantastico )

    $($CopiaArray $1 TMP)
    TMP=( ${TMP[@]} ${esagerato[@]} )
    $($CopiaArray TMP $2)
}

declare -a prima=( Lo scripting di Bash avanzato )
declare -a dopo
```

```
echo "Array iniziale = ${prima[@]}"

Enfatizza prima dopo

echo "Array finale = ${dopo[@]}"

# Troppo esagerato?

echo "Cos'è ${dopo[@]:4:2}?"

declare -a modesto=( ${dopo[@]:0:2} "è" ${dopo[@]:4:2} )
#                      - estrazione di sottostringhe -

echo "Array modesto = ${modesto[@]}"

# Cos'è successo a 'prima' ?

echo "Array iniziale = ${prima[@]}"

exit 0
```

Example 26-10. Ancora sulla concatenazione di array

```
#! /bin/bash
# array-append.bash

# Copyright (c) Michael S. Zick, 2003, Tutti i diritti riservati.
# Licenza: Uso illimitato in qualsiasi forma e per qualsiasi scopo.
# Versione: $ID$
#
# Impaginazione leggermente modificata da M.C.


# Le operazioni degli array sono specifiche di Bash.
# La /bin/sh originaria UNIX non ne possiede di equivalenti.


#  Collagate con una pipe l'output dello script a 'more'
#+ in modo che non scorra completamente sullo schermo.


# Inizializzazione abbreviata.
declare -a array1=( zero1 uno1 due1 )
# Inizializzazione dettagliata ([1] non viene definito).
declare -a array2=( [0]=zero2 [2]=due2 [3]=tre2 )

echo
echo "- Conferma che l'array è stato inizializzato per singolo elemento. -"
echo "Numero di elementi: 4"        # Codificato a scopo illustrativo.
for (( i = 0 ; i < 4 ; i++ ))
do
```

```
    echo "Elemento [$i]: ${array2[$i]}"
done
# Vedi anche il codice d’esempio più generale in basics-reviewed.bash.


declare -a dest

# Combina (accodando) i due array in un terzo.
echo
echo ’Condizioni: Senza quoting, IFS preimpostato, operatore Array-intero’
echo ’- Elementi non definiti assenti, indici non mantenuti. -’
# Gli elementi non definiti non esistono; non vengono inseriti.

dest=( ${array1[@]} ${array2[@]} )
# dest=${array1[@]}${array2[@]}     # Risultati strani, probabilmente un bug.

# Ora visualizziamo il risultato.
echo
echo "- - Verifica dell’accodamento dell’array - -"
cnt=${#dest[@]}

echo "Numero di elementi: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Elemento [$i]: ${dest[$i]}"
done

# (Doppio) Assegnamento di un intero array ad un elemento di un altro array.
dest[0]=${array1[@]}
dest[1]=${array2[@]}

# Visualizzazione del risultato.
echo
echo "- - Verifica dell’array modificato - -"
cnt=${#dest[@]}

echo "Numero di elementi: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Elemento [$i]: ${dest[$i]}"
done

# Esame del secondo elemento modificato.
echo
echo ’- - Riassegnazione e visualizzazione del secondo elemento - -’

declare -a subArray=${dest[1]}
cnt=${#subArray[@]}

echo "Numero di elementi: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Elemento [$i]: ${subArray[$i]}"
```

```
done

#  L’assegnamento di un intero array ad un singolo elemento
#+ di un altro array, utilizzando la notazione ’=${ ... }’,
#+ ha trasformato l’array da assegnare in una stringa,
#+ con gli elementi separati da uno spazio (il primo carattere di IFS).

# Se gli elementi d’origine non avessero contenuto degli spazi . . .
# Se l’array d’origine non fosse stato inizializzato in modo dettagliato . . .
# Allora come risultato si sarebbe ottenuto la struttura dell’array d’origine.

# Ripristino con il secondo elemento modificato.
echo
echo "- - Visualizzazione dell’elemento ripristinato - -"

declare -a subArray=( ${dest[1]} )
cnt=${#subArray[@]}

echo "Numero di elementi: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Elemento [$i]: ${subArray[$i]}"
done
echo ’- - Non fate affidamento su questo comportamento. - -’
echo ’- - Potrebbe divergere nelle versioni di Bash - -’
echo ’- - precedenti alla 2.05b - -’

# MSZ: Mi scuso per qualsiasi confusa spiegazione fatta in precedenza.

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.

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

```
#!/bin/bash
# bubble.sh: Ordinamento a bolle.

# Ricordo l’algoritmo dell’ordinamento a bolle. In questa particolare versione.
..

#  Ad ogni passaggio successivo lungo l’array che deve essere ordinato,
#+ vengono confrontati due elementi adiacenti e scambiati se non ordinati.
#  Al termine del primo passaggio, l’elemento "più pesante" è sprofondato
#+ nell’ultima posizione dell’array. Al termine del secondo passaggio, il
#+ rimanente elemento "più pesante" si trova al penultimo posto. E così via.
#+ Questo significa che ogni successivo passaggio deve attraversare una
#+ porzione minore di array. Noterete, quindi, un aumento della velocità
#+ di visualizzazione dopo ogni passaggio.
```

```
scambio()
{
  # Scambia due membri dell'array.
  local temp=${Paesi[$1]} #  Variabile per la memorizzazione temporanea
                          #+ dell'elemento che deve essere scambiato.
  Paesi[$1]=${Paesi[$2]}
  Paesi[$2]=$temp

  return
}

declare -a Paesi  #  Dichiara l'array,
                  #+ in questo caso facoltativo perché viene inizializzato
  #+ successivamente.

#  È consentito suddividere l'inizializzazione di un array su più righe
#+ utilizzando il carattere di escape (\)?
#  Sì.

Paesi=(Olanda Ucraina Zaire Turchia Russia Yemen Siria \
Brasile Argentina Nicaragua Giappone Messico Venezuela Grecia Inghilterra \
Israele Peru Canada Oman Danimarca Galles Francia Kenya \
Xanadu Qatar Liechtenstein Ungheria)

# "Xanadu" è il luogo mitico dove, secondo Coleridge,
#+"Kubla Khan fece un duomo di delizia fabbricare".


clear                  # Pulisce lo schermo prima di iniziare l'elaborazione.

echo "0: ${Paesi[*]}"  # Elenca l'intero array al passaggio 0.

numero_di_elementi=${#Paesi[@]}
let "confronti = $numero_di_elementi - 1"

conto=1 # Numero di passaggi.

while [ "$confronti" -gt 0 ]               # Inizio del ciclo esterno
do

  indice=0  # L'indice viene azzerato all'inizio di ogni passaggio.

  while [ "$indice" -lt "$confronti" ]      # Inizio del ciclo interno
  do
    if [ ${Paesi[$indice]} \> ${Paesi[`expr $indice + 1`]} ]
    #  Se non ordinato...
    #  Ricordo che \> è l'operatore di confronto ASCII
    #+ usato all'interno delle parantesi quadre singole.

    #  if [[ ${Paesi[$indice]} > ${Paesi[`expr $indice + 1`]} ]]
    #+ anche in questa forma.
    then
```

```
        scambio $indice ‘expr $indice + 1‘  # Scambio.
    fi
    let "indice += 1"
  done # Fine del ciclo interno


let "confronti -= 1" #  Poiché l’elemento "più pesante" si è depositato in
     #+ fondo, è necessario un confronto in meno ad ogni
     #+ passaggio.

echo
echo "$conto: ${Paesi[@]}"  #  Visualizza la situazione dell’array al termine
    #+ di ogni passaggio.
echo
let "conto += 1"            # Incrementa il conteggio dei passaggi.

done                        # Fine del ciclo esterno
                            # Completato.

exit 0
```

--

È possibile annidare degli array in altri array?

```
#!/bin/bash
# Array annidato.

# Esempio fornito da Michael Zick.

UnArray=( $(ls --inode --ignore-backups --almost-all \
--directory --full-time --color=none --time=status \
--sort=time -l ${PWD} ) )  # Comandi e opzioni.

# Gli spazi sono significativi . . . quindi non si deve usare il quoting.

SubArray=( ${UnArray[@]:11:1}  ${UnArray[@]:6:5} )
# Array formato da due elementi, ognuno dei quali è a sua volta un array.

echo "Directory corrente e data dell’ultima modifica:"
echo "${SubArray[@]}"

exit 0
```

--

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

Example 26-12. Array annidati e referenziazioni indirette

```
#!/bin/bash
# embedded-arrays.sh
# Array annidati e referenziazione indiretta.

# Script di Dennis Leeuw.
# Usato con il permesso dell'autore.
# Modificato dall'autore di questo documento.


ARRAY1=(
        VAR1_1=valore11
        VAR1_2=valore12
        VAR1_3=valore13
)

ARRAY2=(
        VARIABILE="test"
        STRINGA="VAR1=valore1 VAR2=valore2 VAR3=valore3"
        ARRAY21=${ARRAY1[*]}
)       # L'ARRAY1 viene inserito in questo secondo array.

function visualizza () {
        PREC_IFS="$IFS"
        IFS=$'\n'       #  Per visualizzare ogni elemento dell'array
                        #+ su una riga diversa.
        TEST1="ARRAY2[*]"
        local ${!TEST1} # Provate a vedere cosa succede cancellando questa riga.
        #  Referenziazione indiretta.
#  Questo rende i componenti di $TEST1
#+ accessibili alla funzione.


        #  A questo punto, vediamo cosa abbiamo fatto.
        echo
        echo "\$TEST1 = $TEST1"       #  Solo il nome della variabile.
        echo; echo
        echo "{\$TEST1} = ${!TEST1}"  #  Contenuto della variabile.
                                      #  Questo è ciò che fa la
                                      #+ referenziazione indiretta.
        echo
        echo "-----------------------------------------"; echo
        echo


        # Visualizza la variabile
        echo "Variabile VARIABILE: $VARIABILE"

        # Visualizza un elemento stringa
        IFS="$PREC_IFS"
        TEST2="STRINGA[*]"
        local ${!TEST2}      # Referenziazione indiretta (come prima).
```

```
        echo "Elemento stringa VAR2: $VAR2 da STRINGA"

        # Visualizza un elemento dell'array
        TEST2="ARRAY21[*]"
        local ${!TEST2}      # Referenziazione indiretta (come prima).
        echo "Elemento VAR1_1 dell'array: $VAR1_1 da ARRAY21"
}

visualizza
echo

exit 0

#   Come ha fa notare l'autore dello script,
#+ "lo script può facilmente essere espanso per ottenere gli hash
#+  anche nella shell bash."
#   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.

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

```
#!/bin/bash
# sieve.sh

# Crivello di Eratostene
# Antico algoritmo per la ricerca di numeri primi.

# L'esecuzione è di due ordini di grandezza
# più lenta dell'equivalente programma scritto in C.

LIMITE_INFERIORE=1       # Si inizia da 1.
LIMITE_SUPERIORE=1000    # Fino a 1000.
# (Potete impostarlo ad un valore più alto...  se avete tempo a disposizione.)

PRIMO=1
NON_PRIMO=0

let META=LIMITE_SUPERIORE/2
# Ottimizzazione:
# È necessario verificare solamente la metà dei numeri.


declare -a Primi
# Primi[] è un array.


inizializza ()
{
```

```
# Inizializza l’array.

i=$LIMITE_INFERIORE
until [ "$i" -gt "$LIMITE_SUPERIORE" ]
do
  Primi[i]=$PRIMO
  let "i += 1"
done
# Assumiamo che tutti gli elementi dell’array siano colpevoli (primi)
# finché non verrà provata la loro innocenza (non primi).
}

visualizza_primi ()
{
# Visualizza gli elementi dell’array Primi[] contrassegnati come primi.

i=$LIMITE_INFERIORE

until [ "$i" -gt "$LIMITE_SUPERIORE" ]
do

  if [ "${Primi[i]}" -eq "$PRIMO" ]
  then
    printf "%8d" $i
    # 8 spazi per numero danno delle belle ed uniformi colonne.
  fi

  let "i += 1"

done

}

vaglia () # Identifica i numeri non primi.
{

let i=$LIMITE_INFERIORE+1
# Sappiamo che 1 è primo, quindi iniziamo da 2.

until [ "$i" -gt "$LIMITE_SUPERIORE" ]
do

if [ "${Primi[i]}" -eq "$PRIMO" ]
#  Non si preoccupa di vagliare i numeri già verificati (contrassegnati come
#+ non-primi).
then

  t=$i

  while [ "$t" -le "$LIMITE_SUPERIORE" ]
  do
    let "t += $i "
    Primi[t]=$NON_PRIMO
```

```
    # Segna come non-primi tutti i multipli.
  done

fi

  let "i += 1"
done


}


# Invoca le funzioni sequenzialmente.
inizializza
vaglia
visualizza_primi
# Questa è quella che si chiama programmazione strutturata.

echo

exit 0



# ----------------------------------------------- #
# Il codice oltre la riga precedente non viene eseguito.

# Questa versione migliorata del Crivello, di Stephane Chazelas,
# esegue il compito un po' più velocemente.

#  Si deve invocare con un argomento da riga di comando (il limite dei
#+ numeri primi).

LIMITE_SUPERIORE=$1            # Da riga di comando.
let META=LIMITE_SUPERIORE/2    # Metà del numero massimo.

Primi=( '' $(seq $LIMITE_SUPERIORE) )

i=1
until (( ( i += 1 ) > META ))  #  È sufficiente verificare solo la metà dei
                               #+ numeri.
do
  if [[ -n $Primi[i] ]]
  then
    t=$i
    until (( ( t += i ) > LIMITE_SUPERIORE ))
    do
      Primi[t]=
    done
  fi
done
echo ${Primi[*]}
```

```
exit 0
```

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

--

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

Example 26-14. Simulare uno stack push-down

```
#!/bin/bash
# stack.sh: simulazione di uno stack push-down

#  Simile ad uno stack di CPU, lo stack  push-down registra i dati
#+ sequenzialmente, ma li rilascia in ordine inverso, last-in first-out
#+ (l'ultimo inserito è il primo prelevato).

BP=100            # Base Pointer (puntatore alla base) dello stack (array).
                  # Inizio dall'elemento 100.

SP=$BP            # Stack Pointer (puntatore allo stack).
                  # Viene inizializzato alla "base" (fondo) dello stack.

Dato=             # Contenuto di una locazione dello stack .
                  #  Deve essere una variabile locale,
                  #+ a causa della limitazione del valore di ritorno di
  #+ una funzione.

declare -a stack


push()            # Pone un dato sullo stack.
{
if [ -z "$1" ]    # Niente da immettere?
then
  return
fi

let "SP -= 1"     # Riposiziona lo stack pointer.
stack[$SP]=$1

return
}

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
#   + nello stack di più dati con un'unica chiamata.

# 2)  Modificate la funzione "pop()" in modo che consenta di prelevare
#   + dallo stack più dati con un'unica chiamata.

# 3)  Utilizzando questo script come base di partenza, scrivete un programma
#   + per una calcolatrice a 4 funzioni basate sullo stack.

exit 0

#  N.d.T. - Si è preferito lasciare inalterati i termini, in quanto
#+ appartenenti al linguaggio di programmazione Assembly. La traduzione è
#+ 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.

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

```
#!/bin/bash

# I celebri "numeri Q" di Douglas Hofstadter

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), per n>2

#  È una successione di interi "caotica" con comportamento strano e
#+ non prevedibile.
# I primi 20 numeri della serie sono:
# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

#  Vedi il libro di Hofstadter, "Goedel, Escher, Bach: un'Eterna Ghirlanda
#+ Brillante", p. 149, ff. (Ed. italiana Adelphi - terza edizione - settembre
#+ 1985 [N.d.T.])


LIMITE=100      # Numero di termini da calcolare
AMPIEZZARIGA=20 # Numero di termini visualizzati per ogni riga

Q[1]=1          # I primi due numeri della serie sono 1.
Q[2]=1

echo
echo "Numeri Q [$LIMITE termini]:"
echo -n "${Q[1]} "              # Visualizza i primi due termini.
echo -n "${Q[2]} "

for ((n=3; n <= $LIMITE; n++))  # ciclo con condizione in stile C.
do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  per n>2
```

```
#  È necessario suddividere l'espressione in termini intermedi,
#  perché Bash non è in grado di gestire molto bene la matematica complessa
#+ degli array.

  let "n1 = $n - 1"        # n-1
  let "n2 = $n - 2"        # n-2

  t0=‘expr $n - ${Q[n1]}‘  # n - Q[n-1]
  t1=‘expr $n - ${Q[n2]}‘  # n - Q[n-2]

  T0=${Q[t0]}              # Q[n - Q[n-1]]
  T1=${Q[t1]}              # Q[n - Q[n-2]]

Q[n]=‘expr $T0 + $T1‘      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
echo -n "${Q[n]} "

if [ ‘expr $n % $AMPIEZZARIGA‘ -eq 0 ]    # Ordina l'output.
then   #    modulo
  echo # Suddivide le righe in blocchi ordinati.
fi

done

echo

exit 0

#  Questa è un'implementazione iterativa dei numeri Q.
#  L'implementazione più intuitiva, che utilizza la ricorsività, è lasciata
#+ come esercizio.
# 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.

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

```
#!/bin/bash
# twodim.sh: Simulazione di un array bidimensionale.

# Un array bidimensionale registra le righe sequenzialmente.

Righe=5
Colonne=5

declare -a alfa      # alfa [Righe] [Colonne];
                     # Dichiarazione non necessaria. Perché?

inizializza_alfa ()
{
local rc=0
```

```
local indice


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
do     # Se preferite, potete utilizzare simboli differenti.
  local riga=‘expr $rc / $Colonne‘
  local colonna=‘expr $rc % $Righe‘
  let "indice = $riga * $Righe + $colonna"
  alfa[$indice]=$i
# alfa[$riga][$colonna]
  let "rc += 1"
done

#  Sarebbe stato più semplice
#   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 )
#+ ma, per così dire, si sarebbe perso il "gusto" dell’array bi-dimensionale.
}

visualizza_alfa ()
{
local riga=0
local indice

echo

while [ "$riga" -lt "$Righe" ] # Visualizza in ordine di precedenza di riga -
do                             # variano le colonne
                               # mentre la riga (ciclo esterno) non cambia.
  local colonna=0

  while [ "$colonna" -lt "$Colonne" ]
  do
   let "indice = $riga * $Righe + $colonna"
   echo -n "${alfa[indice]} "  # alfa[$riga][$colonna]
   let "colonna += 1"
  done

  let "riga += 1"
  echo

done

# L’analogo più semplice è
#   echo ${alfa[*]} | xargs -n $Colonne

echo
}

filtra ()     # Elimina gli indici negativi dell’array.
{

echo -n "  "  # Fornisce l’inclinazione.
              # Spiegate perché.
```

```
if [[ "$1" -ge 0 &&  "$1" -lt "$Righe" && "$2" -ge 0 && "$2" -lt "$Colonne" ]]
then
 let "indice = $1 * $Righe + $2"
 # Ora lo visualizza ruotato.
 echo -n " ${alfa[indice]}"
 #           alfa[$riga][$colonna]
fi

}




ruota ()  #  Ruota l'array di 45 gradi
{         #+ (facendo "perno" sul suo angolo inferiore sinistro).
local riga
local colonna

for (( riga = Righe; riga > -Righe; riga-- ))
do          # Passa l'array in senso inverso.

  for (( colonna = 0; colonna < Colonne; colonna++ ))
  do

    if [ "$riga" -ge 0 ]
    then
        let "t1 = $colonna - $riga"
        let "t2 = $colonna"
    else
       let "t1 = $colonna"
       let "t2 = $colonna + $riga"
    fi

    filtra $t1 $t2   # Elimina gli indici negativi dell'array.
  done

  echo; echo

done

#  La rotazione è ispirata agli esempi (pp. 143-146) presenti in
#+ "Advanced C Programming on the IBM PC", di Herbert Mayer
#+ (vedi bibliografia).

}


#--------------- E ora, che lo spettacolo inizi.-------------#
inizializza_alfa  # Inizializza l'array.
visualizza_alfa   # Lo visualizza.
ruota             # Lo ruota di 45 gradi in senso antiorario.
#------------------------------------------------------------#
```

```
# Si tratta di una simulazione piuttosto macchinosa, per non dire cervellotica.
#
# Esercizi:
# ---------
# 1)  Riscrivete le funzioni di inizializzazione e visualizzazione
#   + in maniera più intuitiva ed elegante.
#
# 2)  Illustrate come operano le funzioni di rotazione dell'array.
#     Suggerimento: pensate alle implicazioni di una indicizzazione
#     inversa dell'array.

exit 0
```

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 array bidimensionale, vedi Example A-11.

Notes

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

Chapter 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 Appendix H 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.

Notes

1. Questo non è valido per **csh**, **tcsh** e per tutte le altre shell non imparentate o non derivanti dalla classica shell Bourne (**sh**).

Chapter 28. /dev e /proc

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 è 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 Example 13-6 e Example 13-5.

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

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
```

Gli script di shell possono ricavare dati da alcuni dei file presenti in `/proc`. [3]

```
bash$ cat /proc/filesystems | grep iso9660
        iso9660
```

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

```
CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )

if [ $CPU = Pentium ]
then
  esegui_dei_comandi
  ...
else
  esegui_altri_comandi
  ...
fi
```

La directory `/proc` contiene delle sottodirectory con strani nomi numerici. Ognuno di questi nomi traccia l'ID di processo dei processi correntemente in esecuzione. All'interno di ognuna di queste sottodirectory, vi è un certo numero di file contenenti utili informazioni sui corrispondenti processi. 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.

Example 28-1. Trovare il processo associato al PID

```
#!/bin/bash
#  pid-identifier.sh: Fornisce il percorso completo del processo associato al
#+ pid.

ARGNUM=1  # Numero di argomenti attesi dallo script.
E_ERR_ARG=65
E_ERR_PID=66
E_ERR_PROCESSO=67
E_ERR_PERMESSO=68
FILEPROC=exe

if [ $# -ne $ARGNUM ]
then
  echo "Utilizzo: 'basename $0' numero PID" >&2  # Messaggio d'errore >stderr.
  exit $E_ERR_ARG
fi

pidnum=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
#  Controlla il pid nell'elenco di "ps", campo nr.1.
#  Quindi si accerta che sia il processo effettivo, non quello invocato dallo
#+ script stesso.
# L'ultimo "grep $1" scarta questa possibilità.
if [ -z "$pidnum" ]  #  Se, anche dopo il filtraggio, il risultato è una
                     #+ stringa di lunghezza zero,
then                 #  significa che nessun processo in esecuzione
                     #+ corrisponde al pid dato.
  echo "Il processo non è in esecuzione."
  exit $E_ERR_PROCESSO
fi

# In alternativa:
```

```
#   if ! ps $1 > /dev/null 2>&1
#   then             # nessun processo in esecuzione corrisponde al pid dato.
#     echo "Il processo non è in esecuzione."
#     exit $E_ERR_PROCESSO
#    fi

# Per semplificare l'intera procedura, si usa "pidof".


if [ ! -r "/proc/$1/$FILEPROC" ]  # Controlla i permessi in lettura.
then
  echo "Il processo $1 è in esecuzione, ma..."
  echo "Non ho il permesso di lettura su /proc/$1/$FILEPROC."
  exit $E_ERR_PERMESSO  #  Un utente ordinario non può accedere ad alcuni
                        #+ 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
```

Example 28-2. 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.


if [ -z "$pidnum" ]   #  Se non c'è il pid, allora il processo non è
                      #+ in esecuzione.
then
  echo "Non connesso."
  exit $NONCONNESSO
else
  echo "Connesso."; echo
fi

while [ true ]       # Ciclo infinito. Qui lo script può essere migliorato.
do

  if [ ! -e "/proc/$pidnum/$NOMEFILEPROC" ]
  # Finché il processo è in esecuzione, esiste il file "status".
  then
      echo "Disconnesso."
      exit $NONCONNESSO
  fi

netstat -s | grep "packets received"  # Per avere alcune statistiche.
netstat -s | grep "packets delivered"


  sleep $INTERVALLO
  echo; echo

done

exit 0
```

```
# Così com’è, lo script deve essere terminato con Control-C.

#    Esercizi:
#    ---------
#    Migliorate lo script in modo che termini alla pressione del tasto "q".
#    Rendete lo script più amichevole inserendo altre funzionalità
```

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

Notes

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

Chapter 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`.

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

Sopprimere lo `stderr` (da Example 12-2).

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

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

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

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

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

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):

Example 29-1. Evitare i cookie

```
if [ -f ~/.netscape/cookies ]  # Se esiste, lo cancella.
then
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
#  Tutti i cookie vengono ora spediti nel buco nero, invece di essere salvati
#+ su disco.
```

Usi di `/dev/zero`

Come `/dev/null`, anche `/dev/zero` è uno pseudo file, ma in realtà contiene null (zero numerici, 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.

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

```
#!/bin/bash

# Creare un file di swap.
# Questo script deve essere eseguito da root.

UID_ROOT=0         # Root ha $UID 0.
E_ERR_UTENTE=65    # Non root?

FILE=/swap
DIMENSIONEBLOCCO=1024
BLOCCHIMIN=40
SUCCESSO=0

if [ "$UID" -ne "$UID_ROOT" ]
then
  echo; echo "Devi essere root per eseguire questo script."; echo
  exit $E_ERR_UTENTE
fi


blocchi=${1:-$BLOCCHIMIN}    #  Imposta a 40 blocchi il valore predefinito, se
                             #+ non viene specificato diversamente da riga di
                             #+ comando.
# Equivale al seguente blocco di codice.
# ----------------------------------------------------
# if [ -n "$1" ]
# then
#   blocchi=$1
# else
#   blocchi=$BLOCCHIMIN
```

```
# fi
# ------------------------------------------------


if [ "$blocchi" -lt $BLOCCHIMIN ]
then
  blocchi=$BLOCCHIMIN       # La dimensione deve essere di almeno 40 blocchi.
fi


echo "Creazione di un file di swap della dimensione di $bloccchi blocchi (KB)."
dd if=/dev/zero of=$FILE bs=$DIMENSIONEBLOCCO count=$blocchi  #  Pone il file a
                                                              #+ zero.

mkswap $FILE $blocchi             # Lo designa come file di swap.
swapon $FILE                      # Attiva il file di swap.

echo "Il file di swap è stato creato ed attivato."

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 Example 13-6) o per la cancellazione di sicurezza di un file (vedi Example 12-43).

Example 29-3. Creare un ramdisk

```
#!/bin/bash
# ramdisk.sh

#  Un "ramdisk" è un segmento della memoria RAM
#+ che si comporta come se fosse un filesystem.
#  Presenta il vantaggio di un accesso velocissimo (tempo di lettura/scrittura)
#  Svantaggi: volatilità, perdita di dati al riavvio o in caso di mancanza di
#+ corrente elettrica, meno RAM disponibile al sistema.
#
#  Cos’ha di buono un ramdisk?
#  Tenere una serie di dati di grandi dimensioni, come una tabella o un
#+ dizionario, su un ramdisk ne velocizza la consultazione, perché l’accesso
#+ alla memoria è molto più veloce di un accesso al disco.


E_NON_ROOT=70                 # Deve essere eseguito da root.
NOME_ROOT=root

MOUNTPT=/mnt/ramdisk
DIMENSIONE=2000               # 2K blocchi (modificare in base alle esigenze)
DIMENSIONEBLOCCO=1024         # 1K (1024 byte)
DISPOSITIVO=/dev/ram0         # Primo dispositivo ram

nomeutente=‘id -nu‘
if [ "$nomeutente" != "$NOME_ROOT" ]
then
```

```
  echo "Devi essere root per eseguire \"‘basename $0‘\"."
  exit $E_NON_ROOT
fi

if [ ! -d "$MOUNTPT" ]         #  Verifica se già esiste il punto di mount,
then                           #+ in modo che non ci sia un errore se lo script
  mkdir $MOUNTPT               #+ viene eseguito più volte.
fi

dd if=/dev/zero of=$DISPOSITIVO count=$DIMENSIONE bs=$DIMENSIONEBLOCCO
                               #  Pone il dispositivo RAM a zero.
mke2fs $DISPOSITIVO            #  Crea, su di esso, un filesystem di tipo ext2.
mount $DISPOSITIVO $MOUNTPT    #  Lo monta.
chmod 777 $MOUNTPT             #  Così un utente ordinario può accedere al
                               #+ ramdisk.
                               #  Tuttavia, si deve essere root per smontarlo.

echo "\"$MOUNTPT\" ora è disponibile all’uso."
#  Il ramdisk è accessibile, per la registrazione di file, anche ad un utente
#+ ordinario.

#  Attenzione, il ramdisk è volatile e il contenuto viene perso
#+ in caso di riavvio o mancanza di corrente.
#  Copiate tutto quello che volete salvare in una directory regolare.

# Dopo un riavvio, rieseguite questo script per reimpostare il ramdisk.
# Rifare il mount di /mnt/ramdisk senza gli altri passaggi è inutile.

exit 0
```

Chapter 30. Debugging

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.

Example 30-1. Uno script errato

```
#!/bin/bash
# ex74.sh

# Questo è uno script errato.
# Ma dove sarà mai l'errore?

a=37

if [$a -gt 27 ]
then
  echo $a
fi

exit 0
```

Output dello script:

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

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

Example 30-2. Parola chiave mancante

```
#!/bin/bash
# missing-keyword.sh: Che messaggio d'errore verrà generato?

for a in 1 2 3
do
  echo "$a"
# done     #  La necessaria parola chiave 'done', alla riga 7,
           #+ è stata commentata.

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 dei commenti presenti nello script.

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

Example 30-3. test24, un altro script errato

```
#!/bin/bash

#  Si suppone che questo script possa cancellare tutti i file della
#+ directory corrente i cui nomi contengono degli spazi.
#  Non funziona.
#  Perché?


bruttonome=‘ls | grep ’ ’‘

# echo "$bruttonome"

rm "$bruttonome"

exit 0
```

Si cerchi di scoprire cos’è andato storto in Example 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.

```
# Metodi corretti per cancellare i file i cui nomi contengono spazi.
rm *\ *
rm *" "*
rm *’ ’*
# 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).

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

```
#!/bin/bash
# assert.sh

assert ()                 #  Se la condizione è falsa,
{                         #+ esce dallo script con un messaggio d'errore.
  E_ERR_PARAM=98
  E_ASSERT_FALLITA=99


  if [ -z "$2" ]          # Non sono stati passati abbastanza parametri.
  then
    return $E_ERR_PARAM   # Non fa niente.
  fi

  numriga=$2

  if [ ! $1 ]
  then
    echo "Assert \"$1\" fallita:"
    echo "File \"$0\", riga $numriga"
    exit $E_ASSERT_FALLITA
  # else
  #   return
  #   e continua l'esecuzione dello script.
  fi
}


a=5
b=4
condizione="$a -lt $b"    #  Messaggio d'errore ed uscita dallo script.
                          #  Provate ad impostare "condizione" con
```

```
                            #+ qualcos'altro, e vedete cosa succede.

assert "$condizione" $LINENO
# La parte restante dello script verrà eseguita solo se "assert" non fallisce.


# Alcuni comandi.
# ...
echo "Questo enunciato viene visualizzato solo se \"assert\" non fallisce."
# ...
# Alcuni altri comandi.

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.

Note: 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.
```

Example 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.


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.
```

Example 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
#+ (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.

trap ’rm -f $FILETEMP; exit $INTERRUPT_UTENTE’; TERM INT
# Cancella il file temporaneo se lo script viene interrotto con un control-c.

echo

while [ $TRUE ]  # Ciclo infinito.
do
  tail -$CONTROLLA_RIGHE $FILELOG> $FILETEMP
  #  Salva le ultime 100 righe del file di log di sistema nel file
  #+ temporaneo. Necessario, dal momento che i kernel più
  #+ recenti generano molti messaggi di log durante la fase di avvio.
  ricerca=‘grep $PAROLACHIAVE $FILETEMP‘
  # Verifica la presenza della frase "IP address",
  # che indica che il collegamento è riuscito.

  if [ ! -z "$ricerca" ] #  Sono necessari gli apici per la possibile
                         #+ presenza di spazi.
  then
     echo "Collegato"
     rm -f $FILETEMP     # Cancella il file temporaneo.
     exit $COLLEGATO
  else
```

```
      echo -n "."        # L'opzione -n di echo sopprime il ritorno a capo,
                         # così si ottengono righe continue di punti.
  fi

  sleep 1
done


#  Nota: se sostituite la variabile PAROLACHIAVE con "Exit",
#+ potete usare questo script per segnalare, se si è collegati,
#+ uno scollegamento inaspettato.

#  Esercizio: Modificate lo script per ottenere quanto suggerito nella
#+ nota precedente, rendendolo anche più elegante.

exit 0


# Nick Drage ha suggerito un metodo alternativo:

while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connesso" && exit 0
  echo -n "."   # Visualizza dei punti (.....) finché si è connessi.
  sleep 2
done

# Problema: Può non bastare premere Control-C per terminare il processo.
#           (I punti continuano ad essere visualizzati.)
# Esercizio: Risolvetelo.



# Stephane Chazelas ha un'altra alternativa ancora:

INTERVALLO=1

while ! tail -1 "$FILELOG" | grep -q "$PAROLACHIAVE"
do echo -n .
   sleep $INTERVALLO
done
echo "Connesso"

# Esercizio: Discutete i punti di forza e i punti deboli
#            di ognuno di questi differenti approcci.
```

Note: 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.

Example 30-7. Tracciare una variabile

```
#!/bin/bash

trap ’echo "TRACCIA-VARIABILE> \$variabile = \"$variabile\""’ DEBUG
# Visualizza il valore di $variabile dopo l’esecuzione di ogni comando.

variabile=29;

echo "La \"\$variabile\" è stata inizializzata a $variabile."

let "variabile *= 3"
echo "\"\$variabile\" è stata moltiplicata per 3."

#  Il costrutto "trap ’comandi’ DEBUG" diventa molto utile
#+ nel contesto di uno script complesso, dove collocare molti
#+ enunciati "echo $variabile" si rivela goffo oltre che una perdita
#+ di tempo.

# Grazie, Stephane Chazelas per la puntualizzazione.

exit 0
```

Note: `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.

```
trap ” 2  #  Il segnale 2 è Control-C, che ora è disabilitato.
comando
comando
comando
trap 2     # Riabilita Control-C
```

Notes

1. Il Bash debugger (http://bashdb.sourceforge.net) di Rocky Bernstein colma, in parte, questa lacuna.
2. Convenzionalmente, il *`segnale 0`* è assegnato a exit.

Chapter 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.

```
#!/bin/bash

set -o verbose
# Visualizza tutti i comandi prima della loro esecuzione.
```

```
#!/bin/bash

set -v
# Identico effetto del precedente.
```

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

```
#!/bin/bash

set -o verbose
# Abilitata la visualizzazione dei comandi.
comando
...
comando

set +o verbose
# Visualizzazione dei comandi disabilitata.
comando
# Non visualizzato.


set -v
# Visualizzazione dei comandi abilitata.
comando
...
comando

set +v
# Visualizzazione dei comandi disabilitata.
comando
```

```
exit 0
```

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

```
#!/bin/bash -x
#
# Corpo dello script.
```

È anche possibile abilitare le opzioni di 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`).

Table 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 **...**
`-f`	noglob	Disabilita l'espansione dei nomi dei file (globbing)
`-i`	interactive	Lo script viene eseguito in modalità *interattiva*
`-p`	privileged	Lo script viene eseguito come "suid" (attenzione!)
`-r`	restricted	Lo script viene eseguito in modalità *ristretta* (vedi Chapter 21).

Abbreviazione	Nome	Effetto
`-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
`-e`	errexit	Lo script abortisce al primo errore (quando l'exit status di un comando è diverso da zero)
`-n`	noexec	Legge i comandi dello script, ma non li esegue (controllo di sintassi)
`-s`	stdin	Legge i comandi dallo `stdin`
`-t`	(nessuno)	Esce dopo il primo comando
`-`	(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.

Chapter 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 di variabili, parole o caratteri riservati.

```
case=valore0       # Causa problemi.
23skidoo=valore1   # Ancora problemi.
#  I nomi di variabili che iniziano con una cifra sono riservati alla shell.
#  Sostituite con  _23skidoo=valore1. I nomi che iniziano con un
#+ underscore (trattino di sottolineatura) vanno bene.

# Tuttavia...      usare il solo underscore non funziona.
_=25
echo $_            #  $_ è la variabile speciale impostata
                   #+ all'ultimo argomento dell'ultimo comando.

xyz((!*=valore2    # Provoca seri problemi.
```

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

```
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.

```
fa_qualcosa ()
{
  echo "Questa funzione fa qualcosa con \"$1\"."
}

fa_qualcosa=fa_qualcosa

fa_qualcosa fa_qualcosa

# Tutto questo è consentito, ma estremamente disorientante.
```

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

```
var1 = 23   # corretto 'var1=23'.
# Nella riga precedente, Bash cerca di eseguire il comando "var1"
# con gli argomenti "=" e "23".

let c = $a - $b   # corretto 'let c=$a-$b' o 'let "c = $a - $b"'.
```

```
if [ $a -le 5]    # corretto if [ $a -le 5 ]   .
# if [ "$a" -le 5 ]   ancora meglio.
# [[ $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.

```
#!/bin/bash

echo "var_non_inizializzata = $var_non_inizializzata"
# 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.

```
if [ "$a" = 273 ]      # $a è un intero o una stringa?
if [ "$a" -eq 273 ]    # $a è un intero.

# Talvolta è possibile scambiare -eq con = senza alcuna conseguenza.
# Tuttavia...


a=273.0   # Non è un intero.

if [ "$a" = 273 ]
then
  echo "Il confronto ha funzionato."
else
  echo "Il confronto non ha funzionato."
fi    # Il confronto non ha funzionato.

# Stessa cosa con   a=" 273"  e a="0273".


# Allo stesso modo, si hanno problemi ad usare "-eq" con valori non interi.

if [ "$a" -eq 273.0 ]
then
  echo "a = $a’
fi  # Esce con un messaggio d’errore.
# test.sh: [: 273.0: integer expression expected
```

Non confondere gli operatori per il confronto di interi con quelli di confronto di stringhe.

```
#!/bin/bash
# bad-op.sh

numero=1
```

```
while [ "$numero" < 5 ]    #  Sbagliato! Dovrebbe essere
                           #+ while [ "numero" -lt 5 ]
do
  echo -n "$numero "
  let "numero += 1"
done

# Il tentativo di esecuzione provoca un messaggio d'errore:
# bad-op.sh: 5: No such file or directory
```

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 Example 7-6, Example 16-4 e Example 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.

```
comando1 2> - | comando2  #  Il tentativo di redirigere l'output
                          #+ d'errore di comando 1 con una pipe...
                          #    ...non funziona.

comando1 2>& - | comando2 # Altrettanto inutile.

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.

```
#!/bin/bash

versione_minima=2
# Dal momento che Chet Ramey sta costantemente aggiungendo funzionalità a Bash,
# si può impostare $versione_minima a 2.XX, o ad altro valore appropriato.
E_ERR_VERSIONE=80

if [ "$BASH_VERSION" \< "$versione_minima" ]
then
  echo "Questo script funziona solo con Bash, versione"
  echo "$versione_minima o superiore."
  echo "Se ne consiglia caldamente l'aggiornamento."
  exit $E_ERR_VERSIONE
fi

...
```

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 Example 23-5.

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.

```
#!/bin/bash

echo "Si parte"

unix2dos $0    # lo script si trasforma nel formato DOS.
chmod 755 $0   # Viene ripristinato il permesso di esecuzione.
               # Il comando 'unix2dos' elimina i permessi di escuzione.

./$0           # Lo script tenta la riesecuzione.
               # Come file DOS non può più funzionare.

echo "Fine"

exit 0
```

Uno script di shell che inizia con **#!/bin/sh** potrebbe non funzionare 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.

```
QUELLO_CHE_VUOI=/home/bozo
export QUELLO_CHE_VUOI
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.

Example 32-1. I trabocchetti di una Subshell

```
#!/bin/bash
# Le insidie delle variabili di una subshell.

variabile_esterna=esterna
echo
echo "variabile esterna = $variabile_esterna"
echo

(
# Inizio della subshell

echo "variabile esterna nella subshell = $variabile_esterna"
variabile_interna=interna  # Impostata
echo "variabile interna nella subshell = $variabile_interna"
variabile_esterna=interna  # Il valore risulterà cambiato a livello globale?
echo "variabile esterna nella subshell = $variabile_esterna"

# Fine della subshell
)

echo
echo "variabile interna al di fuori della subshell = $variabile_interna"
             # Non impostata.
echo "variabile esterna al di fuori della subshell = $variabile_esterna"
             # Immutata.
echo

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 Example 11-15).

Example 32-2. Concatenare con una pipe l'output di echo a read

```
#!/bin/bash
#  badread.sh:
#  Tentativo di usare 'echo e 'read'
#+ per l'assegnazione non interattiva di variabili.

a=aaa
b=bbb
c=ccc

echo "uno due tre" | read a b c
# Cerca di riassegnare a, b e c.

echo
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
```

```
# Riassegnazione fallita.

# ------------------------------

# Proviamo la seguente alternativa.

var=‘echo "uno due tre"‘
set -- $var
a=$1; b=$2; c=$3

echo "-------"
echo "a = $a"  # a = uno
echo "b = $b"  # b = due
echo "c = $c"  # c = tre
# Riassegnazione riuscita.

# ------------------------------

#  Notate inoltre che echo con ’read’ funziona all’interno di una subshell.
#  Tuttavia, il valore della variabile cambia *solo* in quell’ambito.

a=aaa          # Ripartiamo da capo.
b=bbb
c=ccc

echo; echo
echo "uno due tre" | ( read a b c;
echo "nella subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = uno
# b = due
# c = tre
echo "----------------------"
echo "Fuori dalla subshell: "
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
echo

exit 0
```

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

```
# Problemi nell’uso di una pipe con un ciclo.
# Esempio di Anthony Richardson.


trovato=falso
find $HOME -type f -atime +30 -size 100k |
while true
do
   read f
   echo "$f supera i 100KB e non è stato usato da più di 30 giorni"
   echo "Considerate la possibilità di spostarlo in un archivio."
```

```
   trovato=vero
done

#  In questo caso trovato sarà sempre falso perchè
#+ è stato impostato all'interno di una subshell
if [ $trovato = falso ]
then
   echo "Nessun file da archiviare."
fi

# ====================Ora, il modo corretto:=======================

trovato=falso
for f in $(find $HOME -type f -atime +30 -size 100k)  # Nessuna pipe.
do
   echo "$f supera i 100KB e non è stato usato da più di 30 giorni"
   echo "Considerate la possibilità di spostarlo in un archivio."
   trovato=vero
done

if [ $trovato = falso ]
then
   echo "Nessun file da archiviare."
fi
```

--

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

Notes

1. L’impostazione del bit *suid* dello script stesso non ha alcun effetto.

Chapter 33. Scripting con stile

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.

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

  Si aggiungano intestazioni descrittive agli script e alle funzioni.

  ```
  #!/bin/bash

  #********************************************#
  #                   xyz.sh                   #
  #           scritto da Bozo Bozeman          #
  #               05 luglio 2001               #
  #                                            #
  #      Cancellazione dei file di progetto.   #
  #********************************************#

  ERRDIR=65                       # Directory inesistente.
  dirprogetti=/home/bozo/projects # Directory da cancellare.

  # ------------------------------------------------------------------ #
  # cancella_filep ()                                                  #
  # Cancella tutti i file della directory specificata.                 #
  # Parametro: $directory_indicata                                     #
  # Restituisce: 0 in caso di successo, $ERRDIR se qualcosa va storto. #
  # ------------------------------------------------------------------ #
  cancella_filep ()
  {
    if [ ! -d "$1" ]  # Verifica l’esistenza della directory indicata.
    then
       echo "$1 non è una directory."
       return $ERRDIR
     fi

    rm -f "$1"/*
    return 0   # Successo.
  }
  ```

```
cancella_filep $dirprogetti

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.

```
if [ -f /var/log/messages ]
then
  ...
fi
#  L'anno successivo decidete di cambiare lo script per
#+ verificare /var/log/syslog.
#  È necessario modificare manualmente lo script, un'occorrenza
#+ alla volta, e sperare che tutto funzioni a dovere.

# Un modo migliore:
FILELOG=/var/log/messages  # Basterà cambiare solo questa riga.
if [ -f "$FILELOG" ]
then
  ...
fi
```

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

```
ef='ls -al $nomedir'                 #  Criptico.
elenco_file='ls -al $nomedir'        #  Meglio.


VALMAX=10                            #  I nomi delle costanti in
                                     #+ lettere maiuscole.
while [ "$indice" -le "$VALMAX" ]
...


E_NONTROVATO=75                      #  Costanti dei codici d'errore
                                     #+ in maiuscolo,
                                     #  e con i nomi che iniziano con "E_".
if [ ! -e "$nomefile" ]
then
  echo "Il file $nomefile non è stato trovato."
  exit $E_NONTROVATO
fi


MAIL_DIRECTORY=/var/spool/mail/bozo  #  Lettere maiuscole per le variabili
                                     #+ d'ambiente.
export MAIL_DIRECTORY
```

```
LeggiRisposta ()                    #  Iniziali maiuscole per i nomi di
                                    #+ funzione.
{
  prompt=$1
  echo -n $prompt
  read risposta
  return $risposta
}

LeggiRisposta "Qual'è il tuo numero preferito? "
numero_preferito=$?
echo $numero_preferito


_variabileutente=23                 # Consentito, ma non raccomandato.
#  È preferibile che i nomi delle variabili definite dall'utente non inizino
#+ con un underscore.
#  Meglio lasciarlo per le variabili di sistema.
```

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

  ```
  E_ERR_ARG=65
  ...
  ...
  exit $E_ERR_ARG
  ```

  Vedi anche Appendix D.

- Si suddividano gli script complessi in moduli più semplici. Si faccia uso delle funzioni ogni qual volta se ne presenti l'occasione. Vedi Example 35-4.

- Non si usi un costrutto complesso dove uno più semplice è sufficiente.

  ```
  COMANDO if [ $? -eq 0 ]
  ...
  # Ridondante e non intuitivo.

  if COMANDO
  ...
  # 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

Notes

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

Chapter 34. Miscellanea

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

Tom Duff

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.

```
#!/bin/bash
MIO_PROMPT='$ '
while :
do
  echo -n "$MIO_PROMPT"
  read riga
  eval "$riga"
  done

exit 0

#  Questo script d'esempio e gran parte della spiegazione precedente
#+ 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 Example 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.)

```
if [ -z $PS1 ] # nessun prompt?
then
  # non interattiva
  ...
else
  # interattiva
  ...
fi
```

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

```
case $- in
*i*)    # shell interattiva
;;
*)      # shell non interattiva
;;
# (Cortesia di "UNIX F.A.Q.," 1993)
```

> **Note:** 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.

Example 34-1. Shell wrapper

```
#!/bin/bash

# Questo è un semplice script che rimuove le righe vuote da un file.
# Nessuna verifica d'argomento.
#
```

```
# Sarebbe meglio aggiungere qualcosa come:
# if [ -z "$1" ]
# then
#  echo "Utilizzo: ‘basename $0‘ nome-file"
#  exit 65
# fi


# È uguale a
#    sed -e ’/^$/d’ nomefile
# invocato da riga di comando.

sed -e /^$/d "$1"
#  ’-e’ significa che segue un comando di "editing" (in questo caso opzionale).
#  ’^’ indica l’inizio della riga, ’$’ la fine.
#  Verifica le righe che non contengono nulla tra il loro inizio e la fine,
#+ vale a dire, le righe vuote.
#  ’d’ è il comando di cancellazione.

#  L’uso del quoting per l’argomento consente di
#+ passare nomi di file contenenti spazi e caratteri speciali.

exit 0
```

Example 34-2. Uno shell wrapper leggermente più complesso

```
#!/bin/bash

#  "subst", uno script per sostituire un nome
#+ con un altro all’interno di un file,
#+ es., "subst Smith Jones letter.txt".

ARG=3          # Lo script richiede tre argomenti.
E_ERR_ARG=65   # Numero errato di argomenti passati allo script.

if [ $# -ne "$ARG" ]
# Verifica il numero degli argomenti (è sempre una buona idea).
then
  echo "Utilizzo: ‘basename $0‘ vecchio-nome nuovo-nome nomefile"
  exit $E_ERR_ARG
fi

vecchio_nome=$1
nuovo_nome=$2

if [ -f "$3" ]
then
  nome_file=$3
else
  echo "Il file \"$3\" non esiste."
  exit $E_ERR_ARG
fi
```

```
#  Ecco dove il lavoro principale viene svolto.

# ------------------------------------------------
sed -e "s/$vecchio_nome/$nuovo_nome/g" $nome_file
# ------------------------------------------------

#  's' è, naturalmente, il comando sed di sostituzione,
#+ e /modello/ invoca la ricerca di corrispondenza.
#  L'opzione "g", o globale, provoca la sostituzione di *tutte*
#+ le occorrenze di $vecchio_nome in ogni riga, non solamente nella prima.
#  Leggete testi riguardanti 'sed' per una spiegazione più approfondita.

exit 0    # Lo script invocato con successo restituisce 0.
```

Example 34-3. Uno shell wrapper attorno ad uno script awk

```
#!/bin/bash

# Aggiunge la colonna specificata (di numeri) nel file indicato.

ARG=2
E_ERR_ARG=65

if [ $# -ne "$ARG" ] #  Verifica il corretto nr. di argomenti da riga
                     #+ di comando.
then
   echo "Utilizzo: `basename $0` nomefile numero-colonna"
   exit $E_ERR_ARG
fi

nomefile=$1
numero_colonna=$2

# Il passaggio di variabili di shell ad awk, che è parte dello script
#+ stesso, è un po' delicato.
# Vedete la documentazione awk per maggiori dettagli.

# Uno script awk che occupa più righe viene invocato con   awk ' ..... '


# Inizio dello script awk.
# ------------------------------
awk '

{ totale += $'"${numero_colonna}"'
}
END {
      print totale
}

' "$nomefile"
```

```
# ----------------------------
# Fine dello script awk.


#   Potrebbe non essere sicuro passare variabili di shell a uno script awk
#   incorporato, così Stephane Chazelas propone la seguente alternativa:
#   --------------------------------------
#   awk -v numero_colonna="$numero_colonna" '
#   { totale += $numero_colonna
#   }
#   END {
#       print totale
#   }' "$nomefile"
#   --------------------------------------


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).

Example 34-4. Perl inserito in uno script Bash

```
#!/bin/bash

# I comandi shell possono precedere lo script Perl.
echo "Questa riga precede lo script Perl inserito in \"$0\"."
echo "============================================================="

perl -e 'print "Questo è lo script Perl che è stato inserito.\n";'
# Come sed, anche Perl usa l'opzione "-e".

echo "============================================================="
echo "Comunque, lo script può contenere anche comandi di shell e di sistema."

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.

Example 34-5. Script Bash e Perl combinati

```
#!/bin/bash
# bashandperl.sh

echo "Saluti dalla parte Bash dello script."
# Qui possono seguire altri comandi Bash.

exit 0
```

```
# Fine della parte Bash dello script.

# ======================================================

#!/usr/bin/perl
# Questa parte dello script deve essere invocata con l'opzione -x.

print "Saluti dalla parte Perl dello script.\n";
# Qui possono seguire altri comandi Perl.

# 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.

```
a=8

# Tutti i confronti seguenti si equivalgono.
test "$a" -lt 16 && echo "sì, $a < 16"         #  "lista and"
/bin/test "$a" -lt 16 && echo "sì, $a < 16"
[ "$a" -lt 16 ] && echo "sì, $a < 16"
[[ $a -lt 16 ]] && echo "sì, $a < 16"          #  Non è necessario il quoting
                                               #+ delle variabili presenti in [[ ]] e (( )).
(( a < 16 )) && echo "sì, $a < 16"

città="New York"
# Anche qui, tutti i confronti seguenti si equivalgono.
test "$città" \< Parigi && echo "Sì, Parigi è più grande di $città"
# Più grande in ordine ASCII.
/bin/test "$città" \< Parigi && echo "Sì, Parigi è più grande di $città"
[ "$città" \< Parigi ] && echo "Sì, Parigi è più grande di $città"
[[ $città < Parigi ]] && echo "Sì, Parigi è più grande di $città"
# $città senza quoting.

# Grazie, S.C.
```

34.4. Ricorsività

Può uno script richiamare sé stesso ricorsivamente? Certo.

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

```
#!/bin/bash
# recurse.sh

#  Può uno script richiamare sé stesso ricorsivamente?
#  Sì, ma può essere di qualche uso pratico?
#  (Vedi il successivo.)

INTERVALLO=10
VALMAX=9

i=$RANDOM
let "i %= $INTERVALLO"  #  Genera un numero casuale compreso
                        #+ tra 0 e $INTERVALLO - 1.

if [ "$i" -lt "$VALMAX" ]
then
  echo "i = $i"
  ./$0                  #  Lo script genera ricorsivamente una nuova istanza
                        #+ di sé stesso.
fi                      #  Ogni script figlio fa esattamente la stessa
                        #+ cosa, finché un $i non sia uguale a $VALMAX.

#  L’uso di un ciclo "while", invece della verifica "if/then", provoca problemi.
#  Spiegate perché.

exit 0
```

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

```
#!/bin/bash
# pb.sh: phone book

# Scritto da Rick Boivie e usato con il consenso dell’autore.
# Modifiche effettuate dall’autore del documento.

MINARG=1      # Lo script ha bisogno di almeno un argomento.
FILEDATI=./phonebook
              # Deve esistere un file dati di nome "phonebook".
NOMEPROG=$0
E_NON_ARG=70  # Errore di nessun argomento.

if [ $# -lt $MINARG ]; then
      echo "Utilizzo: "$NOMEPROG" filedati"
      exit $E_NON_ARG
fi
```

```
if [ $# -eq $MINARG ]; then
      grep $1 "$FILEDATI"
      # 'grep' visualizza un messaggio d'errore se $FILEDATI non esiste.
else
      ( shift; "$NOMEPROG" $* ) | grep $1
      # Lo script richiama sé stesso ricorsivamente.
fi

exit 0        #  Lo script termina qui.
              #  Va bene mettere dati e commenti senza il #
              #+ oltre questo punto.

# ------------------------------------------------------------------------
# Un estratto del file dati "phonebook":

John Doe        1555 Main St., Baltimore, MD 21228           (410) 222-3333
Mary Moe        9899 Jones Blvd., Warren, NH 03787           (603) 898-3232
Richard Roe     856 E. 7th St., New York, NY 10009           (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009           (212) 444-5678
Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338   (415) 501-1631
# ------------------------------------------------------------------------

$bash pb.sh Roe
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

$bash pb.sh Roe Sam
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

#  Quando vengono passati più argomenti allo script,
#+ viene visualizzata *solo* la/e riga/he contenente tutti gli argomenti.
```

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

```
#!/bin/bash
# usrmnt.sh, scritto da Anthony Richardson
# Utilizzato con il permesso dell'autore.

# utilizzo:    usrmnt.sh
# descrizione: monta un dispositivo, l'utente cho lo invoca deve essere elencato
#              nel gruppo MNTUSERS nel file /etc/sudoers.

# ---------------------------------------------------------------
#  Si tratta dello script usermount che riesegue
#+ se stesso usando sudo.  Un utente con i permessi
#+ appropriati deve digitare semplicemente

#   usermount /dev/fd0 /mnt/floppy

# invece di
```

```
#   sudo usermount /dev/fd0 /mnt/floppy

#  Utilizzo questa tecnica per tutti gli
#+ script sudo perché la trovo conveniente.
# ------------------------------------------------------------

#  Se la variabile SUDO_COMMAND non è impostata, significa che non lo si
#+ sta eseguendo attraverso sudo, che quindi va richiamato. Vengono passati
#+ i veri id utente e di gruppo . . .

if [ -z "$SUDO_COMMAND" ]
then
   mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
   exit 0
fi

# Verrà eseguita questa riga solo se lo si sta eseguendo con sudo
/bin/mount $* -o uid=$mntusr,gid=$grpusr

exit 0

# Note aggiuntive (dell'autore dello script):
# -----------------------------------------------

# 1) Linux consente l'uso dell'opzione "users" nel file /etc/fstab,
#    quindi qualsiasi utente può montare un certo dispositivo.
#    Ma, su un server, è preferibile consentire l'accesso ai dispositivi
#    solo a pochi individui. Trovo che usare sudo dia un maggior
#    controllo.

# 2) Trovo anche che, per ottenere questo risultato, sudo sia più#    conveniente che utilizzare i g

# 3) Questo metodo fornisce, a tutti coloro dotati dei corretti permessi,
#    l'accesso root al comando mount, quindi fate attenzione a chi lo
#    concedete. È possibile ottenere un controllo ancora più
#    preciso utilizzando questa tecnica in differenti script ciascono inerente
#    a mntfloppy, mntcdrom e mntsamba.
```

Caution

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 di primo piano e di sfondo. I file batch DOS usano comunemente i codici di escape ANSI per *colorare* i loro output, e altrettanto possono fare gli script Bash.

Example 34-9. Una rubrica di indirizzi “a colori”

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


clear                                   # Pulisce lo schermo.

echo -n "          "
echo -e '\E[37;44m'"\033[1mElenco Contatti\033[0m"
                                        # Bianco su sfondo blu
echo; echo
echo -e "\033[1mScegliete una delle persone seguenti:\033[0m"
                                        # Grassetto
tput sgr0
echo "(Inserite solo la prima lettera del nome.)"
echo
echo -en '\E[47;34m'"\033[1mE\033[0m"   # Blu
tput sgr0                               # Ripristina i colori "normali."
echo "vans, Roland"                     # "[E]vans, Roland"
echo -en '\E[47;35m'"\033[1mJ\033[0m"   # Magenta
tput sgr0
echo "ones, Mildred"
echo -en '\E[47;32m'"\033[1mS\033[0m"   # Verde
tput sgr0
echo "mith, Julie"
echo -en '\E[47;31m'"\033[1mZ\033[0m"   # Rosso
tput sgr0
echo "ane, Morris"
echo

read persona

case "$persona" in
# Notate l'uso del "quoting" per la variabile.

  "E" | "e" )
  # Accetta sia una lettera maiuscola che una minuscola.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Socio d'affari & vecchio amico"
  ;;

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
```

```
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Fidanzata"
  echo "Compleanno: Feb. 11"
  ;;

# Aggiungete in seguito le informazioni per Smith & Zane.

         * )
   # Opzione preefinita.
   # Anche un input vuoto (è stato premuto il tasto INVIO) viene verificato qui.
   echo
   echo "Non ancora inserito nel database."
  ;;

esac

tput sgr0                                # Ripristina i colori "normali."

echo

exit 0
```

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"
```

> **Note:** 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
```

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

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.

Table 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

Example 34-10. Visualizzare testo colorato

```
#!/bin/bash
# color-echo.sh: Visualizza messaggi colorati.

# Modificate lo script secondo le vostre necessità.
# Più facile che codificare i colori.

nero='\E[30;47m'
rosso='\E[31;47m'
verde='\E[32;47m'
giallo='\E[33;47m'
blu='\E[34;47m'
magenta='\E[35;47m'
cyan='\E[36;47m'
bianco='\E[37;47m'
```

```
alias Reset="tput sgr0"      #  Ripristina gli attributi di testo normali
                             #+ senza pulire lo schermo.


cecho ()                     # Colora-echo.
                             # Argomento $1 = messaggio
                             # Argomento $2 = colore
{
local msg_default="Non è stato passato nessun messaggio."
                             #  Veramente, non ci sarebbe bisogno di una
                             #+ variabile locale.

messaggio=${1:-$msg_default} #  Imposta al messaggio predefinito se non ne
                             #+ viene fornito alcuno.
colore=${2:-$nero}           #  Il colore preimpostato è il nero, se
                             #+ non ne viene specificato un altro.

  echo -e "$colore"
  echo "$messaggio"
  Reset                      # Ripristina i valori normali.

  return
}


# Ora lo mettiamo alla prova.
# ---------------------------------------------------
cecho "Mi sento triste..." $blu
cecho "Il magenta assomiglia molto al porpora." $magenta
cecho "Sono verde dall'invidia." $verde
cecho "Vedi rosso?" $rosso
cecho "Cyan, più familiarmente noto come acqua." $cyan
cecho "Non è stato passato nessun colore (nero di default)."
       # Omesso l'argomento $colore.
cecho "Il colore passato è \"nullo\" (nero di default)." ""
       # Argomento $colore nullo.
cecho
       # Omessi gli argomenti $messaggio e $colore.
cecho "" ""
       # Argomenti $messaggio e $colore nulli.
# ---------------------------------------------------

echo

exit 0

# Esercizi:
# ---------
# 1) Aggiungete l'attributo "grassetto" alla funzione 'cecho ()'.
# 2) Aggiungete delle opzioni per colorare gli sfondi.
```

> **Caution**
>
> 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/color) semplifica considerevolmente l’uso delle sequenze di escape ANSI. Essa sostituisce i goffi costrutti appena trattati con una sintassi chiara e logica.

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 importante, 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.

```
cat "$file" | grep "$parola"

grep "$parola" "$file"

#  Le due linee di comando hanno un effetto identico, ma la seconda viene
#+ 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 Example 12-33.

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.

  ```
  # Accodate (>>) le righe seguenti alla fine di ogni script di cui
  #+ volete tener traccia.

  date>> $SAVE_FILE      #Data e ora.
  echo $0>> $SAVE_FILE   #Nome dello script.
  echo>> $SAVE_FILE      #Riga bianca di separazione.

  #  Naturalmente, SAVE_FILE deve essere definito ed esportato come
  #+ variabile d'ambiente in ~/.bashrc
  # (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?

  ```
  file=dati.txt
  titolo="***Questa è la riga del titolo del file di testo dati***"

  echo $titolo | cat - $file >$file.nuovo
  # "cat -" concatena lo stdout a $file.
  #  Il risultato finale è
  #+ la creazione di un nuovo file con $titolo aggiunto all'*inizio*.
  ```

  Naturalmente, lo può fare anche sed.

- 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.

  ```
  # LIBRERIA PER SCRIPT
  # ------ ------- ------

  # Nota:
  # Non è presente "#!"
  # Né "codice eseguibile".
  ```

```
# Definizioni di variabili utili

UID_ROOT=0             # Root ha $UID 0.
E_NONROOT=101          # Errore di utente non root.
MAXVALRES=256          # Valore di ritorno massimo (positivo) di una funzione.
SUCCESSO=0
INSUCCESSO=-1


# Funzioni

Utilizzo ()            # Messaggio "Utilizzo:".
{
  if [ -z "$1" ]       # Nessun argomento passato.
  then
     msg=nomefile
  else
     msg=$@
  fi

  echo "Utilizzo: ‘basename $0‘ "$msg""
}


Controlla_root ()      # Controlla se è root ad eseguire lo script.
{                      # Dall’esempio "ex39.sh".
  if [ "$UID" -ne "$UID_ROOT" ]
  then
     echo "Devi essere root per eseguire questo script."
     exit $E_NONROOT
  fi
}


CreaNomeFileTemp ()    # Crea un file temporaneo con nome "unico".
{                      # Dall’esempio "ex51.sh".
  prefisso=temp
  suffisso=‘eval date +%s‘
  Nomefiletemp=$prefisso.$suffisso
}


isalpha2 ()            # Verifica se l’*intera stringa* è formata da
                       #+ caratteri alfabetici.
{                      # Dall’esempio "isalpha.sh".
  [ $# -eq 1 ] || return $INSUCCESSO

  case $1 in
  *[!a-zA-Z]*|"") return $INSUCCESSO;;
  *) return $SUCCESSO;;
  esac                 # Grazie, S.C.
}
```

```
abs ()                          # Valore assoluto.
{                               # Attenzione: Valore di ritorno massimo = 256.
  E_ERR_ARG=-999999

  if [ -z "$1" ]                # È necessario passare un argomento.
  then
     return $E_ERR_ARG          # Ovviamente viene restituito il
                                #+ codice d'errore.
  fi

  if [ "$1" -ge 0 ]             # Se non negativo,
  then                          #
     valass=$1                  # viene preso così com'è.
  else                          # Altrimenti,
     let "valass = (( 0 - $1 ))" # cambia il segno.
  fi

  return $valass
}


in_minuscolo ()        #  Trasforma la/e stringa/he passata/e come argomento/i
{                      #+ in caratteri minuscoli.

  if [ -z "$1" ]       #  Se non viene passato alcun argomento,
  then                 #+ invia un messaggio d'errore
     echo "(null)"     #+ (messaggio d'errore in stile C di puntatore vuoto)
     return            #+ e uscita dalla funzione.
  fi

  echo "$@" | tr A-Z a-z
  # Modifica di tutti gli argomenti passati ($@).

  return

#  Usate la sostituzione di comando per impostare una variabile all'output
#+ della funzione.
#  Per esempio:
#    vecchiavar="unA seRiE di LEtTerE mAiUscoLe e MInusColE MisCHiaTe"
#    nuovavar='in_minuscolo "$vecchiavar"'
#    echo "$nuovavar"  # una serie di lettere maiuscole e minuscole mischiate
#
#  Esercizio: Riscrivete la funzione per modificare le lettere minuscole del/gli
#+ argomento/i passato/i in lettere maiuscole ... in_maiuscolo() [facile].

}
```

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

```
## Attenzione.
```

```
rm -rf *.zzy   ##  Le opzioni "-rf" di "rm" sono molto pericolose,
               ##+ in modo particolare se usate con i caratteri jolly.

#+ Continuazione di riga.
#  Questa è la riga 1
#+ di un commento posto su più righe,
#+ e questa è la riga finale.

#* Nota.

#o Elemento di un elenco.

#> Alternativa.
while [ "$var1" != "fine" ]    #> while test "$var1" != "fine"
```

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

```
#!/bin/bash

BLOCCO_DI_COMMENTO=
#  Provate a impostare la variabile precedente ad un qualsiasi altro valore
#+ ed otterrete una spiacevole sorpresa.

if [ $BLOCCO_DI_COMMENTO ]; then

Commento --
========================================
Questa è una riga di commento.
Questa è un'altra riga di commento.
Questa è un'altra riga ancora di commento.
========================================

echo "Questo messaggio non verrà visualizzato."

I blocchi di commento non generano errori! Whee!

fi

echo "Niente più commenti, prego."

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.

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

```
SUCCESSO=0
E_ERR_INPUT=65

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# Un intero è sia uguale a 0 che non uguale a 0.
# 2>/dev/null sopprime il messaggio d'errore.

if [ $? -ne "$SUCCESSO" ]
then
  echo "Utilizzo: 'basename $0' intero"
  exit $E_ERR_INPUT
fi

let "somma = $1 + 25"             #  Darebbe un errore se $1 non
                                  #+ fosse un intero.
echo "Somma = $somma"

# In questo modo si può verificare qualsiasi variabile,
#+ non solo i parametri passati da riga di comando.

exit 0
```

- L'intervallo 0 - 255, per il valore 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` per poi assegnarlo a una variabile.

  Example 34-11. Uno stratagemma per il valore di ritorno

```
#!/bin/bash
# multiplication.sh

moltiplica ()                   # Moltiplica i parametri passati.
{                               # Accetta un numero variabile di argomenti.

  local prodotto=1

  until [ -z "$1" ]             # Finché ci sono parametri...
  do
    let "prodotto *= $1"
    shift
  done

  echo $prodotto                #  Lo visualizza allo stdout,
}                               #+ poiché verrà assegnato ad una variabile.

molt1=15383; molt2=25211
val1='moltiplica $molt1 $molt2'
echo "$molt1 X $molt2 = $val1"
```

```
                                # 387820813

molt1=25; molt2=5; molt3=20
val2=‘moltiplica $molt1 $molt2 $molt3‘
echo "$molt1 X $molt2 X $molt3 = $val2"
                                # 2500

molt1=188; molt2=37; molt3=25; molt4=47
val3=‘moltiplica $molt1 $molt2 $molt3 $molt4‘
echo "$molt1 X $molt2 X $molt3 X molt4 = $val3"
                                # 8173300

exit 0
```

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

```
car_maiuscolo ()             #  Cambia in maiuscolo il carattere iniziale
{                            #+ di un argomento stringa/he passato.

  stringa0="$@"              # Accetta più argomenti.

  primocar=${stringa0:0:1}   # Primo carattere.
  stringa1=${stringa0:1}     # Parte restante della/e stringa/he.

  PrimoCar=‘echo "$primocar" | tr a-z A-Z‘
                             # Cambia in maiuscolo il primo carattere.

  echo "$PrimoCar$stringa1"  # Visualizza allo stdout.

}

nuovastringa=‘car_maiuscolo "ogni frase dovrebbe iniziare \
con una lettera maiuscola."‘
echo "$nuovastringa"       #  Ogni frase dovrebbe iniziare con una
                           #+ lettera maiuscola.
```

Con questo sistema una funzione può “restituire” più valori.

Example 34-12. Un ulteriore stratagemma per il valore di “ritorno”

```
#!/bin/bash
# sum-product.sh
# Una funzione può "restituire" più di un valore.

somma_e_prodotto ()   #  Calcola sia la somma che il prodotto degli
                      #+ argomenti passati.
{
  echo $(( $1 + $2 )) $(( $1 * $2 ))
# Visualizza allo stdout ogni valore calcolato, separato da uno spazio.
}
```

```
echo
echo "Inserisci il primo numero"
read primo

echo
echo "Inserisci il secondo numero"
read secondo
echo

valres=‘somma_e_prodotto $primo $secondo‘    # Assegna l’output della funzione.
somma=‘echo "$valres" | awk ’{print $1}’‘    # Assegna il primo campo.
prodotto=‘echo "$valres" | awk ’{print $2}’‘ # Assegna il secondo campo.

echo "$primo + $secondo = $somma"
echo "$primo * $secondo = $prodotto"
echo

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.

  Example 34-13. Passaggio e restituzione di array

```
#!/bin/bash
# array-function.sh: Passaggio di un array a una funzione e...
#                    "restituzione" di un array da una funzione


Passa_Array ()
{
  local array_passato   # Variabile locale.
  array_passato=( ‘echo "$1"‘ )
  echo "${array_passato[@]}"
  #  Elenca tutti gli elementi del nuovo array
  #+ dichiarato e impostato all’interno della funzione.
}


array_originario=( elemento1 elemento2 elemento3 elemento4 elemento5 )

echo
echo "array originario = ${array_originario[@]}"
#                        Elenca tutti gli elementi dell’array originario.
```

```
# Ecco il trucco che consente di passare un array ad una funzione.
# ***********************************
argomento=‘echo ${array_originario[@]}‘
# ***********************************
#  Imposta la variabile
#+ a tutti gli elementi, separati da spazi, dell’array originario.
#
# È da notare che cercare di passare semplicemente l’array non funziona.


# Ed ecco il trucco che permette di ottenere un array come "valore di ritorno".
# ****************************************
array_restituito=( ‘Passa_Array "$argomento"‘ )
# ****************************************
# Assegna l’output ’visualizzato’ della funzione all’array.

echo "array restituito = ${array_restituito[@]}"

echo "============================================================="

#  Ora, altra prova, un tentativo di accesso all’array (per elencarne
#+ gli elementi) dall’esterno della funzione.
Passa_Array "$argomento"

# La funzione, di per sé, elenca l’array, ma...
#+ non è consentito accedere all’array al di fuori della funzione.
echo "Array passato (nella funzione) = ${array_passato[@]}"
# VALORE NULL perché è una variabile locale alla funzione.

echo

exit 0
```

Per un dimostrazione più elaborata di passaggio di array a funzioni, vedi Example 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 Example 10-12 e Example 10-17.
- 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.

  Dall’esempio "wstrings.sh".

  ```
  wlist=‘strings "$1" | tr A-Z a-z | tr ’[:space:]’ Z | \
  tr -cs ’[:alpha:]’ Z | tr -s ’\173-\377’ Z | tr Z ’ ’‘
  ```

Example 34-14. Divertirsi con gli anagrammi

```
#!/bin/bash
# agram.sh: Giocare con gli anagrammi.

# Trova gli anagrammi di...
LETTERE=etaoinshrdlu

anagram "$LETTERE" |   # Trova tutti gli anagrammi delle lettere fornite...
grep '.......' |       # Di almeno 7 lettere,
grep '^is' |           # che iniziano con 'is'
grep -v 's$' |         # nessun plurale
grep -v 'ed$'          # nessun participio passato di verbi
# E' possibile aggiungere molte altre combinazioni di condizioni

#  Usa l'utility "anagram" che fa parte del pacchetto
#+ dizionario "yawl" dell'autore di questo documento.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.tar.gz

exit 0                 # Fine del codice.

bash$ sh agram.sh
islander
isolate
isolead
isotheral
```

Vedi anche Example 28-2, Example 12-19 e Example A-10.

- Si usino gli "here document anonimi" per commentare blocchi di codice ed evitare di dover commentare ogni singola riga con un #. Vedi Example 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.

```
CMD=comando1                    # Scelta primaria.
PianoB=comando2                 # Comando di ripiego.

verifica_comando=$(whatis "$CMD" | grep 'nothing appropriate')
#  Se 'comando1' non viene trovato sul sistema , 'whatis' restituisce
#+ "comando1: nothing appropriate."


if [[ -z "$verifica_comando" ]] # Verifica se il comando è presente.
then
  $CMD opzione1 opzione2        #  Esegue comando1 con le opzioni.
else                            #  Altrimenti,
  $PianoB                       #+ esegue comando2.
fi
```

- 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 (http://www.batse.msfc.nasa.gov/~mallozzi/home/software/xforms/src/widtools-2.0.tgz).

> **Caution**
>
> 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.

Example 34-15. Widget invocati da uno script di shell

```
#!/bin/bash
# dialog.sh: Uso dei widgets 'gdialog'.
# Per l'esecuzione dello script è indispensabile aver installato 'gdialog'.

# Lo script è stato ispirato dal seguente articolo.
#     "Scripting for X Productivity," di Marco Fioretti,
#      LINUX JOURNAL, Numero 113, Settembre 2003, pp. 86-9.
# Grazie a tutti quelli di LJ.


# Errore di input nel box di dialogo.
E_INPUT=65
# Dimensioni dei widgets di visualizzazione e di input.
ALTEZZA=50
LARGHEZZA=60

# Nome del file di output (composto con il nome dello script).
OUTFILE=$0.output

# Visualizza questo script in un widget di testo.
gdialog --title "Visualizzazione: $0" --textbox $0 $ALTEZZA $LARGHEZZA


# Ora, proviamo a salvare l'input in un file.
echo -n "VARIABILE=\"" > $OUTFILE  # Usate il quoting nel caso l'input
                                   #+ contenga degli spazi.
gdialog --title "Input Utente" --inputbox "Prego, inserisci un dato:" \
$ALTEZZA $LARGHEZZA 2>> $OUTFILE


if [ "$?" -eq 0 ]
# È buona pratica controllare l'exit status.
then
```

```
  echo "Eseguito \"box di dialogo\" senza errori."
else
  echo "Errore(i) nell'esecuzione di \"box di dialogo\"."
        # Oppure avete cliccato su "Cancel" invece che su "OK".
  rm $OUTFILE
  exit $E_INPUT
fi


echo -n "\"" >> $OUTFILE           # Virgolette finali alla variabile.
#  Questo comando è stato posto qui in fondo per non confondere
#+ l'exit status precedente.


# Ora, recuperiamo e visualizziamo la variabile.
. $OUTFILE   # 'Include' il file salvato.
echo "La variabile inserita nel \"box di input\" è: "$VARIABILE""

rm $OUTFILE  # Cancellazione del file temporaneo.
             # Alcune applicazioni potrebbero aver bisogno di questo file.

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).

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.

Vari ricercatori dei Bell Labs, e di altri posti, 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. [2]

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.

34.9. Portabilità

Questo libro tratta specificamente dello scripting di shell 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. Anche senza queste precauzioni, però, la maggior parte degli script Bash

funzionano senza alcuna modifica sotto **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 (http://linux.oreillynet.com/pub/a/linux/2002/02/28/caldera.html) di molte delle utility originali UNIX non farà che accelerare 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. (ftp://ftp.cwru.edu/pub/bash/FAQ) per un elenco completo.

34.10. Lo scripting di shell sotto Windows

Anche gli utilizzatori di *quell'altro* SO possono eseguire script di shell simil-UNIX e, quindi, beneficiare di molte delle lezioni di questo libro. Il pacchetto Cygwin (http://sourceware.cygnus.com/cygwin/), di Cygnus, e le MKS utilities (http://www.mkssoftware.com/), di Mortice Kern Associates, aggiungono a Windows le capacità dello scripting di shell.

Notes

1. Naturalmente, ANSI è l'acronimo di American National Standards Institute.
2. Vedi l'articolo di Marius van Oers, Unix Shell Scripting Malware (http://www.virusbtn.com/magazine/archives/200204/malshell.xml) e anche *Denning* in bibliografia.

Chapter 35. Bash, versione 2

La versione corrente di *Bash*, quella che viene eseguita sulle vostre macchine, attualmente è la 2.XX.Y.

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

Questo aggiornamento del classico linguaggio di scripting di Bash, ha aggiunto, tra le altre funzionalità, la gestione degli array, [1] l'espansione di stringa e di parametro, e un metodo migliore per le referenziazioni indirette a variabili.

Example 35-1. Espansione di stringa

```
#!/bin/bash

# Espansione di stringa.
# Introdotta con la versione 2 di Bash.

#  Le stringhe nella forma $'xxx'
#+ consentono l'interpretazione delle sequenze di escape standard.

echo $'Tre segnali acustici \a \a \a'
     # Su alcuni terminali potrebbe venir eseguito un solo segnale acustico.
echo $'Tre form feed \f \f \f'
echo $'10 ritorni a capo \n\n\n\n\n\n\n\n\n\n'
echo $'\102\141\163\150'   # Bash
                           # Valori ottali di ciascun carattere.

exit 0
```

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

```
#!/bin/bash

# Referenziazione indiretta a variabile.
# Possiede alcuni degli attributi delle referenziazioni del C++.


a=lettera_alfabetica
lettera_alfabetica=z

echo "a = $a"           # Referenziazione diretta.

echo "Ora a = ${!a}"    # Referenziazione indiretta.
#  La notazione ${!variabile} è di molto superiore alla vecchia
#+ "eval var1=\$$var2"

echo

t=cella_3
```

```
cella_3=24
echo "t = ${!t}"        # t = 24
cella_3=387
echo "Il valore di t è cambiato in ${!t}"    # 387

# È utile per il riferimento ai membri di un array o di una tabella,
# o per simulare un array multidimensionale.
# Un’opzione d’indicizzazione sarebbe stata più gradita (sigh).

exit 0
```

Example 35-3. Applicazione di un semplice database, con l’utilizzo della referenziazione indiretta alle variabili

```
#!/bin/bash
#  resistor-inventory.sh
#  Applicazione di un semplice database che utilizza la referenziazione
#+ indiretta alle variabili.

# ============================================================== #
# Dati

B1723_valore=470                              # ohm
B1723_potenzadissip=.25                       # watt
B1723_colori="giallo-viola-marrone"           # colori di codice
B1723_loc=173                                 # posizione
B1723_inventario=78                           # quantità

B1724_valore=1000
B1724_potenzadissip=.25
B1724_colori="marrone-nero-rosso"
B1724_loc=24N
B1724_inventario=243

B1725_valore=10000
B1725_potenzadissip=.25
B1725_colori="marrone-nero-arancione"
B1725_loc=24N
B1725_inventario=89

# ============================================================== #


echo

PS3=’Inserisci il numero di catalogo: ’

echo

select numero_catalogo in "B1723" "B1724" "B1725"
do
  Inv=${numero_catalogo}_inventario
  Val=${numero_catalogo}_valore
```

```
  Pdissip=${numero_catalogo}_potenzadissip
  Loc=${numero_catalogo}_loc
  Codcol=${numero_catalogo}_colori

  echo
  echo "Numero di catalogo $numero_catalogo:"
  echo "In magazzino ci sono ${!Inv} resistori da\
[${!Val} ohm / ${!Pdissip} watt]."
  echo "Si trovano nel contenitore nr. ${!Loc}."
  echo "Il loro colore di codice è \"${!Codcol}\"."

  break
done

echo; echo

# Esercizio:
# ----------
#  Riscrivete lo script utilizzando gli array, al posto della
#+ referenziazione indiretta a variabile.
#  Quale, tra i due, è il metodo più diretto e intuitivo?


# Nota:
# -----
#  Gli script di shell non sono appropriati per le applicazioni di database,
#+ tranne quelle più semplici. Anche in questi casi, però,
#+ bisogna ricorrere ad espedienti e trucchi vari.
#  È molto meglio utilizzare un linguaggio che abbia un
#+ supporto nativo per le strutture, come C++ o Java (o anche Perl).

exit 0
```

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

```
#!/bin/bash
#  Su macchine un po' datate, potrebbe essere necessario invocarlo
#+ con  #!/bin/bash2.

# Carte:
# distribuzione di un mazzo di carte a quattro giocatori.

NONDISTRIBUITA=0
DISTRIBUITA=1

GIÀ_ASSEGNATA=99

LIMITE_INFERIORE=0
LIMITE_SUPERIORE=51
CARTE_PER_SEME=13
CARTE=52
```

```
declare -a Mazzo
declare -a Semi
declare -a Carte
# Sarebbe stato più semplice ed intuitivo
# con un unico array tridimensionale.
# Forse una futura versione di Bash supporterà gli array multidimensionali.


Inizializza_Mazzo ()
{
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.
{
Seme=‘eval date +%s‘
let "Seme %= 32766"
RANDOM=$Seme
}

Da_Carte ()
{
echo

carte_date=0
while [ "$carte_date" -le $LIMITE_SUPERIORE ]
do
  Sceglie_Carta
  t=$?

  if [ "$t" -ne $GIÀ_ASSEGNATA ]
  then
    Determina_Carta $t

    u=$carte_date+1
    # Ritorniamo all’indicizzazione in base 1 (temporaneamente).
    let "u %= $CARTE_PER_SEME"
    if [ "$u" -eq 0 ]   # Costrutto condizionale if/then annidato.
    then
     echo
     echo
    fi
    # Separa i giocatori.

    let "carte_date += 1"
  fi
done

echo

return 0
}


# Programmazione strutturata:
# l’intero programma è stato "modularizzato" con le funzioni.

#================
Seme_Casuale
```

```
Inizializza_Mazzo
Inizializza_Semi
Inizializza_Carte
Da_Carte

exit 0
#================



# Esercizio 1:
# Aggiungete commenti che spieghino completamente lo script.

# Esercizio 2:
# Revisionate lo script per visualizzare la distribuzione ordinata per seme.
# Potete aggiungere altri fronzoli, si vi aggrada.

# Esercizio 3:
# Semplificate e raffinate la logica dello script.
```

Notes

1. Chet Ramey ha promesso gli array associativi (una funzionalità Perl) in una futura release di Bash.

Chapter 36. Note conclusive

36.1. Nota dell’autore

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 sull’argomento? 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...

36.2. A proposito dell’autore

L’autore non rivendica particolari credenziali o qualifiche, tranne il bisogno di scrivere. [2] 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 (http://personal.riverusers.com/~thegrendel/hmw50.zip). Ha inoltre scritto Software-Building HOWTO (http://tldp.org/HOWTO/Software-Building-HOWTO.html). 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 (http://ibiblio.org/pub/Linux/utils/file/cruft-0.2.tar.gz), utility di cifratura one-time pad; mcalc (http://ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz), per il calcolo del piano d’ammortamento di un mutuo; judge (http://ibiblio.org/pub/Linux/games/amusements/judge-1.0.tar.gz), arbitro per partite di Scrabble® e il pacchetto yawl (http://ibiblio.org/pub/Linux/libs/yawl-0.3.tar.gz) 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 moglie e cane, presso una solitaria comunità nel deserto, riconosce il valore della fragilità umana.

36.3. Dove cercare aiuto

L’autore (mailto:thegrendel@theriver.com) 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 (news:comp.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 (http://www.vim.org) di Bram Moolenaar, in modalità SGML.

ii. OpenJade (http://www.netfolder.com/DSSSL/), motore di rendering DSSSL, per la conversione di documenti SGML in altri formati.

iii. I fogli di stile DSSSL di Norman Walsh (http://nwalsh.com/docbook/dsssl/).

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 (mailto:feloy@free.fr) 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 (mailto: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 Example 4-5.

Stephane Chazelas (mailto:stephane_chazelas@yahoo.fr) 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!

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 (mailto:mikaku@fiwix.org) per aver concesso il permesso all'uso del suo bello script tree (Example A-18).

Allo stesso modo, grazie a Michel Charpentier (mailto:charpov@cs.unh.edu) per il consenso all'uso del suo script per la fattorizzazione dc (Example 12-38).

Onore a Noah Friedman (mailto:friedman@prep.ai.mit.edu) per aver permesso l'utilizzo del suo script di funzioni stringa (Example A-19).

Emmanuel Rouat (mailto:emmanuel.rouat@wanadoo.fr) ha suggerito correzioni ed aggiunte sulla sostituzione di comando e sugli alias. Ha anche fornito un esempio molto bello di file `.bashrc` (Appendix H).

Heiner Steven (mailto:heiner.steven@odn.de) ha gentilmente acconsentito all'uso del suo script per la conversione di base, Example 12-34. Ha, inoltre, eseguito numerose correzioni e fornito utili suggerimenti. Un grazie particolare.

Rick Boivie ha fornito il delizioso script ricorsivo *pb.sh* (Example 34-7) e suggerito miglioramenti per le prestazioni dello script *monthlypmt.sh* (Example 12-33).

Florian Wisser mi ha chiarito alcune sfumature della verifica delle stringhe (vedi Example 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 esempio 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 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, Paulo Marcel Coelho Aragao, Lee Bigelow, Wayne Pollock, "jipe", Emilio Conti, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Sebastien Godard, Björn Eriksson, "nyal," John MacDonald, Joshua Tschida, Manfred Schwarb, Amit Singh, Bill Gradwohl, e David Lawyer (egli stesso autore di 4 HOWTO).

La mia gratitudine a Chet Ramey (mailto:chet@po.cwru.edu) 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 (http://www.tldp.org). 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, Red Hat, la Free Software Foundation (http://www.fsf.org) 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.

Notes

1. Trattasi della celebre tecnica dello "spremere come un limone".
2. Chi può, fa. Chi non può... prende un MCSE (Microsoft Certified Systems Engineer - Attestato di Tecnico di Sistemi Certificato Microsoft [N.d.T.]).

Bibliography

Edited by Peter Denning, *Computers Under Attack: Intruders, Worms, and Viruses,* ACM Press, 1990, 0-201-53067-8.

Questo compendio contiene due articoli su script di shell virus.

*

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

Per poter dispiegare pienamente la potenza dello scripting di shell bisogna avere almeno una certa familiarità, seppur superficiale, con **sed** e **awk**. Questa è la guida standard. Comprende un'eccellente spiegazione delle "espressioni regolari". È un libro da leggere.

*

Aeleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

Questo eccellente manuale per l'amministrazione di sistema contiene una parte, piuttosto buona, dedicata alle basi dello scripting di shell che interessano gli amministratori di sistema e svolge un buon lavoro nella spiegazione degli script di installazione e d'avvio. Dopo una lunga attesa, finalmente è stata pubblicata la terza edizione di questo classico.

*

Stephen Kochan and Patrick Woods, *Unix Shell Programming*, Hayden, 1990, 067248448X.

La guida di riferimento standard sebbene, attualmente, un po' datata.

*

Neil Matthew and Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Ottima e approfondita trattazione dei diversi linguaggi di programmazione per Linux, contenente un capitolo veramente consistente sullo scripting di shell.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Eccellente analisi di algoritmi e regole generali di programmazione.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Ottime informazioni sullo scripting di shell, con esempi ed una breve introduzione a Tcl e Perl.

*

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

Discreto manuale che rappresenta un valido sforzo per l'introduzione alla shell, ma difetta nell'esposizione di argomenti attinenti alla programmazione, nonché di un sufficiente numero di esempi.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

Utilissima guida di riferimento tascabile, sebbene tralasci di trattare le funzionalità specifiche di Bash.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1-56592-260-3.

Contiene un paio di dettagliate sezioni, con articoli approfonditi, sulla programmazione di shell, ma insufficiente come manuale. Sulle espressioni regolari, riporta molto del succitato libro di Dougherty e Robbins.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

Un tesoro ritrovato di idee e formule per esplorare, con il computer, molte curiosità matematiche.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

Il classico manuale di metodi per la soluzione di problemi (cioè, algoritmi).

*

Chet Ramey and Brian Fox, *The GNU Bash Reference Manual (http://www.network-theory.co.uk/bash/manual/)*, Network Theory Ltd, 2003, 0-9541617-7-7.

Questo manuale è la guida di riferimento finale per Bash GNU. Gli autori, Chet Ramey e Brian Fox, sono gli sviluppatori di Bash GNU. Per ogni copia venduta l'editore devolve un dollaro alla Free Software Foundation.

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

Un'eccellente guida di riferimento tascabile per Bash (da non dimenticare mai a casa). Un affare a $ 4.95, ma che è anche possibile scaricare on-line (http://www.ssc.com/ssc/bash/) in formato pdf.

*

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

In assoluto il miglior manuale e guida di riferimento su **awk**. La versione elettronica, libera, di questo libro fa parte della documentazione di **awk**, mentre le copie stampate sono disponibili presso O'Reilly and Associates.

Questo libro è servito d'ispirazione all'autore di questo documento.

*

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

Questo libro, ben scritto, contiene ottimi suggerimenti sullo scripting di shell.

*

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

Testo introduttivo molto dettagliato e di piacevole lettura sull'amministrazione del sistema Linux.

Il libro è disponibile in forma stampata o on-line (http://rute.sourceforge.net/).

*

Ellen Siever and lo staff della O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

La migliore, e più completa, guida di riferimento ai comandi Linux, con una sezione dedicata a Bash.

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

Raccolta, su CD ROM, di sette libri su UNIX, tra cui *UNIX Power Tools*, *Sed and Awk* e *Learning the Korn Shell*. Una serie completa di tutti i manuali e guide di riferimento UNIX, di cui dovreste aver bisogno, a circa 130 dollari. Acquistatela, anche se questo significa far debiti e non pagare l'affitto.

*

I libri O’Reilly su Perl (in realtà, tutti i libri O’Reilly).

Fioretti, Marco, “Scripting for X Productivity,” LINUX JOURNAL, numero 113, settembre, 2003, pp. 86-9.

I begli articoli di Ben Okopnik *introductory Bash scripting* nei numeri 53, 54, 55, 57 e 59 di Linux Gazette (http://www.linuxgazette.com) e la sua spiegazione su “The Deep, Dark Secrets of Bash” nel numero 56.

Chet Ramey’s *bash - The GNU Shell*, serie in due parti pubblicata nei numeri 3 e 4 di Linux Journal (http://www.linuxjournal.com), Luglio-Agosto 1994.

Bash-Programming-Intro HOWTO (http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html) di Mike G.

UNIX Scripting Universe (http://www.injunea.demon.co.uk/index.htm) di Richard.

Bash F.A.Q. (ftp://ftp.cwru.edu/pub/bash/FAQ) di Chet Ramey.

Shell Corner (http://www.unixreview.com/columns/schaefer/) di Ed Schaefer in Unix Review (http://www.unixreview.com).

Gli script di shell d’esempio presso Lucc’s Shell Scripts (http://alge.anart.no/linux/scripts/).

Gli script di shell d’esempio presso SHELLdorado (http://www.shelldorado.com).

Gli script di shell d’esempio al sito di Noah Friedman (http://clri6f.gsi.de/gnu/bash-2.01/examples/scripts.noah/).

Shell Programming Stuff (http://steve-parker.org/sh/sh.shtml) di Steve Parker.

Gli script di shell d'esempio presso SourceForge Snippet Library - shell scrips (http://sourceforge.net/snippet/browse.php?by=lang&lang=7).

Bash-Prompt HOWTO (http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/) di Giles Orr.

I bellissimi manuali su **sed**, **awk** e le espressioni regolari presso The UNIX Grymoire (http://www.grymoire.com/Unix/index.html).

La Sed resources page (http://www.student.northpark.edu/pemente/sed/), di Eric Pement.

Il manuale di riferimento (http://sunsite.ualberta.ca/Documentation/Gnu/gawk-3.0.6/gawk.html) di **gawk** GNU (**gawk** è la versione GNU estesa di **awk** disponibile sui sistemi Linux e BSD).

Il Groff tutorial (http://www.cs.pdx.edu/~trent/gnu/groff/groff.html), di Trent Fisher.

Printing-Usage HOWTO (http://www.tldp.org/HOWTO/Printing-Usage-HOWTO.html) di Mark Komarinski.

Vi è dell'ottimo materiale sulla redirezione I/O nel capitolo 10 della documentazione di textutils (http://sunsite.ualberta.ca/Documentation/Gnu/textutils-2.0/html_chapter/textutils_10.html) sul sito della University of Alberta (http://sunsite.ualberta.ca/Documentation).

Rick Hohensee (mailto:humbubba@smarty.smart.net) ha scritto osimpa (ftp://ftp.gwdg.de/pub/linux/install/clienux/interim/osimpa.tgz), assembler i386 implementato interamente con script Bash.

Aurelio Marinho Jargas ha scritto Regular expression wizard (http://txt2regex.sf.net). Ha inoltre scritto un libro (http://guia-er.sf.net) molto istruttivo sulle Espressioni Regolari, in portoghese.

Ben Tomkins (mailto:brtompkins@comcast.net) ha creato Bash Navigator (http://bashnavigator.sourceforge.net), strumento di gestione delle directory.

Rocky Bernstein sta procedendo nello sviluppo di un “maturo e collaudato” debugger (http://bashdb.sourceforge.net) per Bash.

L’eccellente "Bash Reference Manual" di Chet Ramey e Brian Fox, distribuito come parte del pacchetto "bash-2-doc"(disponibile in formato rpm). Si vedano in particolare gli istruttivi script d’esempio presenti nel pacchetto.

Il newsgroup comp.os.unix.shell (news:comp.unix.shell).

Le pagine di manuale di **bash** e **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **bc**, **xargs**. La documentazione texinfo di **bash**, **dd**, **m4**, **gawk** e **sed**.

Appendix A. Script aggiuntivi

Questi script, sebbene non siano stati inseriti nel testo del documento, illustrano alcune interessanti tecniche di programmazione di shell. Sono anche utili. Ci si diverta ad analizzarli e a eseguirli.

Example A-1. manview: Visualizzare pagine di manuale ben ordinate

```
#!/bin/bash
# manview.sh: Formats the source of a man page for viewing.

#  This script is useful when writing man page source.
#  It lets you look at the intermediate results on the fly
#+ while working on it.

E_WRONGARGS=65

if [ -z "$1" ]
then
  echo "Usage: ‘basename $0‘ filename"
  exit $E_WRONGARGS
fi

# ---------------------------
groff -Tascii -man $1 | less
# From the man page for groff.
# ---------------------------

#  If the man page includes tables and/or equations,
#+ then the above code will barf.
#  The following line can handle such cases.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
#   Thanks, S.C.

exit 0
```

Example A-2. mailformat: Impaginare un messaggio e-mail

```
#!/bin/bash
# mail-format.sh: Format e-mail messages.

# Gets rid of carets, tabs, also fold excessively long lines.

# =================================================================
#                 Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]  # Correct number of arguments passed to script?
```

```
then
  echo "Usage: ‘basename $0‘ filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # Check if file exists.
then
    file_name=$1
else
    echo "File \"$1\" does not exist."
    exit $E_NOFILE
fi
# ===================================================================

MAXWIDTH=70          # Width to fold long lines to.

#  Delete carets and tabs at beginning of lines,
#+ then fold lines to $MAXWIDTH characters.
sed ’
s/^>//
s/^  *>//
s/^  *//
s/ *//
’ $1 | fold -s --width=$MAXWIDTH
          # -s option to "fold" breaks lines at whitespace, if possible.

#  This script was inspired by an article in a well-known trade journal
#+ extolling a 164K Windows utility with similar functionality.
#
#  An nice set of text processing utilities and an efficient
#+ scripting language provide an alternative to bloated executables.

exit 0
```

Example A-3. rn: Una semplice utility per rinominare un file

Questo script è una modifica di Example 12-16.

```
#! /bin/bash
#
# Very simpleminded filename "rename" utility (based on "lowercase.sh").
#
#  The "ren" utility, by Vladimir Lanin (lanin@csd2.nyu.edu),
#+ does a much better job of this.


ARGS=2
E_BADARGS=65
ONE=1                     # For getting singular/plural right (see below).

if [ $# -ne "$ARGS" ]
then
  echo "Usage: ‘basename $0‘ old-pattern new-pattern"
```

```
  # As in "rn gif jpg", which renames all gif files in working directory to jpg.
  exit $E_BADARGS
fi

number=0                 # Keeps track of how many files actually renamed.


for filename in *$1*     #Traverse all matching files in directory.
do
   if [ -f "$filename" ]  # If finds match...
   then
     fname=‘basename $filename‘            # Strip off path.
     n=‘echo $fname | sed -e "s/$1/$2/"‘   # Substitute new for old in filename.
     mv $fname $n                          # Rename.
     let "number += 1"
   fi
done

if [ "$number" -eq "$ONE" ]               # For correct grammar.
then
 echo "$number file renamed."
else
 echo "$number files renamed."
fi

exit 0


# Exercises:
# ---------
# What type of files will this not work on?
# How can this be fixed?
#
#  Rewrite this script to process all the files in a directory
#+ containing spaces in their names, and to rename them,
#+ substituting an underscore for each space.
```

Example A-4. blank-rename: rinomina file i cui nomi contengono spazi

È una versione ancor più semplice dello script precedente

```
#! /bin/bash
# blank-rename.sh
#
# Substitutes underscores for blanks in all the filenames in a directory.

ONE=1                    # For getting singular/plural right (see below).
number=0                 # Keeps track of how many files actually renamed.
FOUND=0                  # Successful return value.

for filename in *        #Traverse all files in directory.
do
     echo "$filename" | grep -q " "         #  Check whether filename
```

```
     if [ $? -eq $FOUND ]                  #+ contains space(s).
     then
       fname=$filename                     # Strip off path.
       n=‘echo $fname | sed -e "s/ /_/g"‘   # Substitute underscore for blank.
       mv "$fname" "$n"                    # Do the actual renaming.
       let "number += 1"
     fi
done

if [ "$number" -eq "$ONE" ]                # For correct grammar.
then
 echo "$number file renamed."
else
 echo "$number files renamed."
fi

exit 0
```

Example A-5. encryptedpw: Upload a un sito ftp utilizzando una password criptata localmente

```
#!/bin/bash

# Example "ex72.sh" modified to use encrypted password.

#  Note that this is still rather insecure,
#+ since the decrypted password is sent in the clear.
#  Use something like "ssh" if this is a concern.

E_BADARGS=65

if [ -z "$1" ]
then
  echo "Usage: ‘basename $0‘ filename"
  exit $E_BADARGS
fi

Username=bozo           # Change to suit.
pword=/home/bozo/secret/password_encrypted.file
# File containing encrypted password.

Filename=‘basename $1‘  # Strips pathname out of file name

Server="XXX"
Directory="YYY"         # Change above to actual server name & directory.


Password=‘cruft <$pword‘          # Decrypt password.
#  Uses the author’s own "cruft" file encryption package,
#+ based on the classic "onetime pad" algorithm,
#+ and obtainable from:
#+ Primary-site:   ftp://ibiblio.org/pub/Linux/utils/file
#+                 cruft-0.2.tar.gz [16k]
```

```
ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
bye
End-Of-Session
# -n option to "ftp" disables auto-logon.
# "bell" rings ’bell’ after each file transfer.

exit 0
```

Example A-6. copy-cd: Copiare un CD di dati

```
#!/bin/bash
# copy-cd.sh: copying a data CD

CDROM=/dev/cdrom                          # CD ROM device
OF=/home/bozo/projects/cdimage.iso        # output file
#       /xxxx/xxxxxxx/                    Change to suit your system.
BLOCKSIZE=2048
SPEED=2                                   # May use higher speed if supported.

echo; echo "Insert source CD, but do *not* mount it."
echo "Press ENTER when ready. "
read ready                                # Wait for input, $ready not used.

echo; echo "Copying the source CD to $OF."
echo "This may take a while. Please be patient."

dd if=$CDROM of=$OF bs=$BLOCKSIZE         # Raw device copy.


echo; echo "Remove data CD."
echo "Insert blank CDR."
echo "Press ENTER when ready. "
read ready                                # Wait for input, $ready not used.

echo "Copying $OF to CDR."

cdrecord -v -isosize speed=$SPEED dev=0,0 $OF
# Uses Joerg Schilling’s "cdrecord" package (see its docs).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html


echo; echo "Done copying $OF to CDR on device $CDROM."

echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.
read answer
```

```
case "$answer" in
[yY]) rm -f $OF
      echo "$OF erased."
      ;;
*)    echo "$OF not erased.";;
esac

echo

# Exercise:
# Change the above "case" statement to also accept "yes" and "Yes" as input.

exit 0
```

Example A-7. Serie di Collatz

```
#!/bin/bash
# collatz.sh

#  The notorious "hailstone" or Collatz series.
#  --------------------------------------------
#  1) Get the integer "seed" from the command line.
#  2) NUMBER <--- seed
#  3) Print NUMBER.
#  4)  If NUMBER is even, divide by 2, or
#  5)+ if odd, multiply by 3 and add 1.
#  6) NUMBER <--- result
#  7) Loop back to step 3 (for specified number of iterations).
#
#  The theory is that every sequence,
#+ no matter how large the initial value,
#+ eventually settles down to repeating "4,2,1..." cycles,
#+ even after fluctuating through a wide range of values.
#
#  This is an instance of an "iterate",
#+ an operation that feeds its output back into the input.
#  Sometimes the result is a "chaotic" series.


MAX_ITERATIONS=200
# For large seed numbers (>32000), increase MAX_ITERATIONS.

h=${1:-$$}                      #  Seed
                                #  Use $PID as seed,
                                #+ if not specified as command-line arg.

echo
echo "C($h) --- $MAX_ITERATIONS Iterations"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
```

```
do

echo -n "$h "
#          ^^^^^
#           tab

  let "remainder = h % 2"
  if [ "$remainder" -eq 0 ]   # Even?
  then
    let "h /= 2"              # Divide by 2.
  else
    let "h = h*3 + 1"         # Multiply by 3 and add 1.
  fi


COLUMNS=10                    # Output 10 values per line.
let "line_break = i % $COLUMNS"
if [ "$line_break" -eq 0 ]
then
  echo
fi

done

echo

#  For more information on this mathematical function,
#+ see "Computers, Pattern, Chaos, and Beauty", by Pickover, p. 185 ff.,
#+ as listed in the bibliography.

exit 0
```

Example A-8. days-between: Calcolo del numero di giorni compresi tra due date

```
#!/bin/bash
# days-between.sh:    Number of days between two dates.
# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY
#
# Note: Script modified to account for changes in Bash 2.05b
#+      that closed the loophole permitting large negative
#+      integer return values.

ARGS=2                # Two command line parameters expected.
E_PARAM_ERR=65        # Param error.

REFYR=1600            # Reference year.
CENTURY=100
DIY=365
ADJ_DIY=367           # Adjusted for leap year + fraction.
MIY=12
DIM=31
LEAPCYCLE=4
```

```
MAXRETVAL=255         #  Largest permissable
                      #+ positive return value from a function.

diff=                 # Declare global variable for date difference.
value=                # Declare global variable for absolute value.
day=                  # Declare globals for day, month, year.
month=
year=


Param_Error ()        # Command line parameters wrong.
{
  echo "Usage: ‘basename $0‘ [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo "       (date must be after 1/3/1600)"
  exit $E_PARAM_ERR
}


Parse_Date ()                 # Parse date from command line params.
{
  month=${1%%/**}
  dm=${1%/**}                 # Day and month.
  day=${dm#*/}
  let "year = ‘basename $1‘"  # Not a filename, but works just the same.
}


check_date ()                 # Checks for invalid date(s) passed.
{
  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -lt "$REFYR" ] && Param_Error
  # Exit script on bad value(s).
  # Uses "or-list / and-list".
  #
  # Exercise: Implement more rigorous date checking.
}


strip_leading_zero () #  Better to strip possible leading zero(s)
{                     #+ from day and/or month
  return ${1#0}       #+ since otherwise Bash will interpret them
}                     #+ as octal values (POSIX.2, sect 2.9.2.1).


day_index ()          # Gauss’ Formula:
{                     # Days from Jan. 3, 1600 to date passed as param.

  day=$1
  month=$2
  year=$3

  let "month = $month - 2"
  if [ "$month" -le 0 ]
```

```
  then
    let "month += 12"
    let "year -= 1"
  fi

  let "year -= $REFYR"
  let "indexyr = $year / $CENTURY"


  let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr + $indexyr/$LEAPCYCLE + $ADJ_DIY*$month/$MIY
  # For an in-depth explanation of this algorithm, see
  # http://home.t-online.de/home/berndt.schwerdtfeger/cal.htm


  echo $Days

}


calculate_difference ()           # Difference between to day indices.
{
  let "diff = $1 - $2"            # Global variable.
}


abs ()                            #  Absolute value
{                                 #  Uses global "value" variable.
  if [ "$1" -lt 0 ]               #  If negative
  then                            #+ then
    let "value = 0 - $1"          #+ change sign,
  else                            #+ else
    let "value = $1"              #+ leave it alone.
  fi
}



if [ $# -ne "$ARGS" ]             # Require two command line params.
then
  Param_Error
fi

Parse_Date $1
check_date $day $month $year      #  See if valid date.

strip_leading_zero $day           #  Remove any leading zeroes
day=$?                            #  on day and/or month.
strip_leading_zero $month
month=$?

let "date1 = ‘day_index $day $month $year‘"
```

```
Parse_Date $2
check_date $day $month $year

strip_leading_zero $day
day=$?
strip_leading_zero $month
month=$?

#let "date2 = ‘day_index $day $month $year‘"
let "date2 = $(day_index $day $month $year)"


calculate_difference $date1 $date2

abs $diff                           # Make sure it’s positive.
diff=$value

echo $diff

exit 0
#  Compare this script with
#+ the implementation of Gauss’ Formula in a C program at:
#     http://buschencrew.hypermart.net/software/datedif
```

Example A-9. Creare un “dizionario”

```
#!/bin/bash
# makedict.sh  [make dictionary]

# Modification of /usr/sbin/mkdict script.
# Original script copyright 1993, by Alec Muffett.
#
#  This modified script included in this document in a manner
#+ consistent with the "LICENSE" document of the "Crack" package
#+ that the original script is a part of.

#  This script processes text files to produce a sorted list
#+ of words found in the files.
#  This may be useful for compiling dictionaries
#+ and for lexicographic research.


E_BADARGS=65

if [ ! -r "$1" ]                    #  Need at least one
then                                #+ valid file argument.
  echo "Usage: $0 files-to-process"
  exit $E_BADARGS
fi


# SORT="sort"                       #  No longer necessary to define options
```

```
                                    #+ to sort. Changed from original script.

cat $* |                            # Contents of specified files to stdout.
        tr A-Z a-z |                # Convert to lowercase.
        tr ' ' '\012' |             # New: change spaces to newlines.
#       tr -cd '\012[a-z][0-9]' |   #  Get rid of everything non-alphanumeric
                                    #+ (original script).
        tr -c '\012a-z'  '\012' |   #  Rather than deleting
                                    #+ now change non-alpha to newlines.
        sort |                      # $SORT options unnecessary now.
        uniq |                      # Remove duplicates.
        grep -v '^#' |              # Delete lines beginning with a hashmark.
        grep -v '^$'                # Delete blank lines.

exit 0
```

Example A-10. Conversione soundex

```
#!/bin/bash
# soundex.sh: Calculate "soundex" code for names

# ======================================================
#        Soundex script
#              by
#         Mendel Cooper
#     thegrendel@theriver.com
#       23 January, 2002
#
#   Placed in the Public Domain.
#
# A slightly different version of this script appeared in
#+ Ed Schaefer's July, 2002 "Shell Corner" column
#+ in "Unix Review" on-line,
#+ http://www.unixreview.com/documents/uni1026336632258/
# ======================================================


ARGCOUNT=1                      # Need name as argument.
E_WRONGARGS=70

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: 'basename $0' name"
  exit $E_WRONGARGS
fi


assign_value ()                 #  Assigns numerical value
{                               #+ to letters of name.

  val1=bfpv                     # 'b,f,p,v' = 1
  val2=cgjkqsxz                 # 'c,g,j,k,q,s,x,z' = 2
```

```
  val3=dt                     #  etc.
  val4=l
  val5=mn
  val6=r

# Exceptionally clever use of 'tr' follows.
# Try to figure out what is going on here.

value=$( echo "$1" \
| tr -d wh \
| tr $val1 1 | tr $val2 2 | tr $val3 3 \
| tr $val4 4 | tr $val5 5 | tr $val6 6 \
| tr -s 123456 \
| tr -d aeiouy )

# Assign letter values.
# Remove duplicate numbers, except when separated by vowels.
# Ignore vowels, except as separators, so delete them last.
# Ignore 'w' and 'h', even as separators, so delete them first.
#
# The above command substitution lays more pipe than a plumber <g>.

}


input_name="$1"
echo
echo "Name = $input_name"


# Change all characters of name input to lowercase.
# ------------------------------------------------
name=$( echo $input_name | tr A-Z a-z )
# ------------------------------------------------
# Just in case argument to script is mixed case.


# Prefix of soundex code: first letter of name.
# --------------------------------------------


char_pos=0                    # Initialize character position.
prefix0=${name:$char_pos:1}
prefix=`echo $prefix0 | tr a-z A-Z`
                              # Uppercase 1st letter of soundex.

let "char_pos += 1"           # Bump character position to 2nd letter of name.
name1=${name:$char_pos}


# ++++++++++++++++++++++++++ Exception Patch ++++++++++++++++++++++++++++++++
#  Now, we run both the input name and the name shifted one char to the right
#+ through the value-assigning function.
```

```
#  If we get the same value out, that means that the first two characters
#+ of the name have the same value assigned, and that one should cancel.
#  However, we also need to test whether the first letter of the name
#+ is a vowel or 'w' or 'h', because otherwise this would bollix things up.

char1=`echo $prefix | tr A-Z a-z`    # First letter of name, lowercased.

assign_value $name
s1=$value
assign_value $name1
s2=$value
assign_value $char1
s3=$value
s3=9$s3                              #  If first letter of name is a vowel
                                     #+ or 'w' or 'h',
                                     #+ then its "value" will be null (unset).
     #+ Therefore, set it to 9, an otherwise
     #+ unused value, which can be tested for.


if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffix=$s2
else
  suffix=${s2:$char_pos}
fi
# ++++++++++++++++++++++ end Exception Patch ++++++++++++++++++++++++++++++++


padding=000                    # Use at most 3 zeroes to pad.


soun=$prefix$suffix$padding    # Pad with zeroes.

MAXLEN=4                       # Truncate to maximum of 4 chars.
soundex=${soun:0:$MAXLEN}

echo "Soundex = $soundex"

echo

#  The soundex code is a method of indexing and classifying names
#+ by grouping together the ones that sound alike.
#  The soundex code for a given name is the first letter of the name,
#+ followed by a calculated three-number code.
#  Similar sounding names should have almost the same soundex codes.

#   Examples:
#   Smith and Smythe both have a "S-530" soundex.
#   Harrison = H-625
#   Hargison = H-622
#   Harriman = H-655
```

```
#  This works out fairly well in practice, but there are numerous anomalies.
#
#
#  The U.S. Census and certain other governmental agencies use soundex,
#  as do genealogical researchers.
#
#  For more information,
#+ see the "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html



# Exercise:
# --------
# Simplify the "Exception Patch" section of this script.

exit 0
```

Example A-11. “Game of Life”

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"

# ##################################################################### #
# This is the Bash script version of John Conway’s "Game of Life".      #
# "Life" is a simple implementation of cellular automata.               #
# --------------------------------------------------------------------- #
# On a rectangular grid, let each "cell" be either "living" or "dead".  #
# Designate a living cell with a dot, and a dead one with a blank space.#
#  Begin with an arbitrarily drawn dot-and-blank grid,                  #
#+ and let this be the starting generation, "generation 0".            #
# Determine each successive generation by the following rules:          #
# 1) Each cell has 8 neighbors, the adjoining cells                     #
#+   left, right, top, bottom, and the 4 diagonals.                    #
#                       123                                           #
#                       4*5                                           #
#                       678                                           #
#                                                                     #
# 2) A living cell with either 2 or 3 living neighbors remains alive.   #
# 3) A dead cell with 3 living neighbors becomes alive (a "birth").     #
SURVIVE=2                                                             #
BIRTH=3                                                               #
# 4) All other cases result in dead cells.                            #
# ##################################################################### #


startfile=gen0   # Read the starting generation from the file "gen0".
                 # Default, if no other file specified when invoking script.
                 #
if [ -n "$1" ]   # Specify another "generation 0" file.
then
  if [ -e "$1" ] # Check for existence.
```

```
  then
    startfile="$1"
  fi
fi


ALIVE1=.
DEAD1=_
                # Represent living and "dead" cells in the start-up file.

#  This script uses a 10 x 10 grid (may be increased,
#+ but a large grid will will cause very slow execution).
ROWS=10
COLS=10

GENERATIONS=10          #  How many generations to cycle through.
                        #  Adjust this upwards,
                        #+ if you have time on your hands.

NONE_ALIVE=80           #  Exit status on premature bailout,
                        #+ if no cells left alive.
TRUE=0
FALSE=1
ALIVE=0
DEAD=1

avar=                   #  Global; holds current generation.
generation=0            # Initialize generation count.

# ==================================================================


let "cells = $ROWS * $COLS"
                        # How many cells.

declare -a initial      # Arrays containing "cells".
declare -a current

display ()
{

alive=0                 # How many cells "alive".
                        # Initially zero.

declare -a arr
arr=( `echo "$1"` )     # Convert passed arg to array.

element_count=${#arr[*]}

local i
local rowcheck

for ((i=0; i<$element_count; i++))
```

```
do

  # Insert newline at end of each row.
  let "rowcheck = $i % ROWS"
  if [ "$rowcheck" -eq 0 ]
  then
    echo                # Newline.
    echo -n "      "    # Indent.
  fi

  cell=${arr[i]}

  if [ "$cell" = . ]
  then
    let "alive += 1"
  fi

  echo -n "$cell" | sed -e 's/_/ /g'
  # Print out array and change underscores to spaces.
done

return

}

IsValid ()                            # Test whether cell coordinate valid.
{

  if [ -z "$1"  -o -z "$2" ]          # Mandatory arguments missing?
  then
    return $FALSE
  fi

local row
local lower_limit=0                   # Disallow negative coordinate.
local upper_limit
local left
local right

let "upper_limit = $ROWS * $COLS - 1" # Total number of cells.


if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
then
  return $FALSE                       # Out of array bounds.
fi

row=$2
let "left = $row * $ROWS"             # Left limit.
let "right = $left + $COLS - 1"       # Right limit.

if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
then
```

```
  return $FALSE                       # Beyond row boundary.
fi

return $TRUE                          # Valid coordinate.

}


IsAlive ()              # Test whether cell is alive.
                        # Takes array, cell number, state of cell as arguments.
{
  GetCount "$1" $2      # Get alive cell count in neighborhood.
  local nhbd=$?


  if [ "$nhbd" -eq "$BIRTH" ]  # Alive in any case.
  then
    return $ALIVE
  fi

  if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
  then                  # Alive only if previously alive.
    return $ALIVE
  fi

  return $DEAD          # Default.

}


GetCount ()             # Count live cells in passed cell's neighborhood.
                        # Two arguments needed:
# $1) variable holding array
# $2) cell number
{
  local cell_number=$2
  local array
  local top
  local center
  local bottom
  local r
  local row
  local i
  local t_top
  local t_cen
  local t_bot
  local count=0
  local ROW_NHBD=3

  array=( `echo "$1"` )

  let "top = $cell_number - $COLS - 1"    # Set up cell neighborhood.
  let "center = $cell_number - 1"
```

```
  let "bottom = $cell_number + $COLS - 1"
  let "r = $cell_number / $ROWS"

  for ((i=0; i<$ROW_NHBD; i++))          # Traverse from left to right.
  do
    let "t_top = $top + $i"
    let "t_cen = $center + $i"
    let "t_bot = $bottom + $i"


    let "row = $r"                       # Count center row of neighborhood.
    IsValid $t_cen $row                  # Valid cell position?
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_cen]} = "$ALIVE1" ] # Is it alive?
      then                               # Yes?
        let "count += 1"                 # Increment count.
      fi
    fi

    let "row = $r - 1"                   # Count top row.
    IsValid $t_top $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_top]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi

    let "row = $r + 1"                   # Count bottom row.
    IsValid $t_bot $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_bot]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi

  done


  if [ ${array[$cell_number]} = "$ALIVE1" ]
  then
    let "count -= 1"        #  Make sure value of tested cell itself
  fi                        #+ is not counted.


  return $count

}
```

```
next_gen ()               # Update generation array.
{

local array
local i=0

array=( ‘echo "$1"‘ )     # Convert passed arg to array.

while [ "$i" -lt "$cells" ]
do
  IsAlive "$1" $i ${array[$i]}   # Is cell alive?
  if [ $? -eq "$ALIVE" ]
  then                           #  If alive, then
    array[$i]=.                  #+ represent the cell as a period.
  else
    array[$i]="_"                #  Otherwise underscore
   fi                            #+ (which will later be converted to space).
  let "i += 1"
done


# let "generation += 1"   # Increment generation count.

# Set variable to pass as parameter to "display" function.
avar=‘echo ${array[@]}‘   # Convert array back to string variable.
display "$avar"           # Display it.
echo; echo
echo "Generation $generation -- $alive alive"

if [ "$alive" -eq 0 ]
then
  echo
  echo "Premature exit: no more cells alive!"
  exit $NONE_ALIVE        #  No point in continuing
fi                        #+ if no live cells.

}


# ==========================================================

# main ()

# Load initial array with contents of startup file.
initial=( ‘cat "$startfile" | sed -e ’/#/d’ | tr -d ’\n’ |\
sed -e ’s/\./\. /g’ -e ’s/_/_ /g’‘ )
# Delete lines containing ’#’ comment character.
# Remove linefeeds and insert space between elements.

clear          # Clear screen.

echo #         Title
echo "========================"
```

```
echo "    $GENERATIONS generations"
echo "           of"
echo "\"Life in the Slow Lane\""
echo "======================="


# -------- Display first generation. --------
Gen0=‘echo ${initial[@]}‘
display "$Gen0"           # Display only.
echo; echo
echo "Generation $generation -- $alive alive"
# ---------------------------------------


let "generation += 1"     # Increment generation count.
echo

# ------- Display second generation. -------
Cur=‘echo ${initial[@]}‘
next_gen "$Cur"          # Update & display.
# ---------------------------------------

let "generation += 1"     # Increment generation count.

# ------ Main loop for displaying subsequent generations ------
while [ "$generation" -le "$GENERATIONS" ]
do
  Cur="$avar"
  next_gen "$Cur"
  let "generation += 1"
done
# ============================================================

echo

exit 0

# ------------------------------------------------------------
# The grid in this script has a "boundary problem".
# The the top, bottom, and sides border on a void of dead cells.
# Exercise: Change the script to have the grid wrap around,
# +         so that the left and right sides will "touch",
# +         as will the top and bottom.
```

Example A-12. File dati per “Game of Life”

```
# This is an example "generation 0" start-up file for "life.sh".
# ------------------------------------------------------------
#  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,
#+ and an underscore (_) for dead ones. We cannot simply use spaces
#+ for dead cells in this file because of a peculiarity in Bash arrays.
#  [Exercise for the reader: explain this.]
```

```
#
# Lines beginning with a '#' are comments, and the script ignores them.
__.__..___
___._._____
_____.___..
_._______.
____.______
..__...___
____._____
___...____
__.._..___
_..___..__

+++
```

I due script seguenti sono di Mark Moraes della University of Toronto. Si veda l'allegato file "Moraes-COPYRIGHT" per quanto riguarda i permessi e le restrizioni.

Example A-13. behead: Togliere le intestazioni dai messaggi di e-mail e di news

```
#! /bin/sh
# Strips off the header from a mail/News message i.e. till the first
# empty line
# Mark Moraes, University of Toronto

# ==> These comments added by author of this document.

if [ $# -eq 0 ]; then
# ==> If no command line args present, then works on file redirected to stdin.
sed -e '1,/^$/d' -e '/^[  ]*$/d'
# --> Delete empty lines and all lines until
# --> first one beginning with white space.
else
# ==> If command line args present, then work on files named.
for i do
sed -e '1,/^$/d' -e '/^[  ]*$/d' $i
# --> Ditto, as above.
done
fi

# ==> Exercise: Add error checking and other options.
# ==>
# ==> Note that the small sed script repeats, except for the arg passed.
# ==> Does it make sense to embed it in a function? Why or why not?
```

Example A-14. ftpget: Scaricare file via ftp

```
#! /bin/sh
# $Id: ftpget.sh,v 1.1.1.1 2003/06/25 22:41:32 giacomo Exp $
# Script to perform batch anonymous ftp. Essentially converts a list of
# of command line arguments into input to ftp.
# Simple, and quick - written as a companion to ftplist
```

```
# -h specifies the remote host (default prep.ai.mit.edu)
# -d specifies the remote directory to cd to - you can provide a sequence
# of -d options - they will be cd'ed to in turn. If the paths are relative,
# make sure you get the sequence right. Be careful with relative paths -
# there are far too many symlinks nowadays.
# (default is the ftp login directory)
# -v turns on the verbose option of ftp, and shows all responses from the
# ftp server.
# -f remotefile[:localfile] gets the remote file into localfile
# -m pattern does an mget with the specified pattern. Remember to quote
# shell characters.
# -c does a local cd to the specified directory
# For example,
#  ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
# -d ../pub/R3/fixes -c ~/fixes -m 'fix*'
# will get xplaces.shar from ~ftp/contrib on expo.lcs.mit.edu, and put it in
# xplaces.sh in the current working directory, and get all fixes from
# ~ftp/pub/R3/fixes and put them in the ~/fixes directory.
# Obviously, the sequence of the options is important, since the equivalent
# commands are executed by ftp in corresponding order
#
# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
# ==> Angle brackets changed to parens, so Docbook won't get indigestion.
#


# ==> These comments added by author of this document.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# ==> Above 2 lines from original script probably superfluous.

TMPFILE=/tmp/ftp.$$
# ==> Creates temp file, using process id of script ($$)
# ==> to construct filename.

SITE=`domainname`.toronto.edu
# ==> 'domainname' similar to 'hostname'
# ==> May rewrite this to parameterize this for general use.

usage="Usage: $0 [-h remotehost] [-d remotedirectory]... [-f remfile:localfile]... \
[-c localdirectory] [-m filepattern] [-v]"
ftpflags="-i -n"
verbflag=
set -f  # So we can use globbing in -m
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
echo $usage
exit 65
fi
shift
trap 'rm -f ${TMPFILE} ; exit' 0 1 2 3 15
echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
```

```
# ==> Added quotes (recommended in complex echoes).
echo binary >> ${TMPFILE}
for i in $*   # ==> Parse command line args.
do
case $i in
-v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
-h) remhost=$2; shift 2;;
-d) echo cd $2 >> ${TMPFILE};
    if [ x${verbflag} != x ]; then
        echo pwd >> ${TMPFILE};
    fi;
    shift 2;;
-c) echo lcd $2 >> ${TMPFILE}; shift 2;;
-m) echo mget "$2" >> ${TMPFILE}; shift 2;;
-f) f1=‘expr "$2" : "\([^:]*\).*"‘; f2=‘expr "$2" : "[^:]*:\(.*\)"‘;
    echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
--) shift; break;;
esac
done
if [ $# -ne 0 ]; then
echo $usage
exit 65   # ==> Changed from "exit 2" to conform with standard.
fi
if [ x${verbflag} != x ]; then
ftpflags="${ftpflags} -v"
fi
if [ x${remhost} = x ]; then
remhost=prep.ai.mit.edu
# ==> Rewrite to match your favorite ftp site.
fi
echo quit >> ${TMPFILE}
# ==> All commands saved in tempfile.

ftp ${ftpflags} ${remhost} < ${TMPFILE}
# ==> Now, tempfile batch processed by ftp.

rm -f ${TMPFILE}
# ==> Finally, tempfile deleted (you may wish to copy it to a logfile).


# ==> Exercises:
# ==> ---------
# ==> 1) Add error checking.
# ==> 2) Add bells & whistles.

+
```

Antek Sawicki ha fornito lo script seguente che fa un uso molto intelligente degli operatori di sostituzione di parametro discussi in Section 9.3.

Example A-15. password: Generare password casuali di 8 caratteri

```
#!/bin/bash
# May need to be invoked with  #!/bin/bash2  on older machines.
#
# Random password generator for bash 2.x by Antek Sawicki <tenox@tenox.tc>,
# who generously gave permission to the document author to use it here.
#
# ==> Comments added by document author ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
LENGTH="8"
# ==> May change ’LENGTH’ for longer password, of course.


while [ "${n:=1}" -le "$LENGTH" ]
# ==> Recall that := is "default substitution" operator.
# ==> So, if ’n’ has not been initialized, set it to 1.
do
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
# ==> Very clever, but tricky.

# ==> Starting from the innermost nesting...
# ==> ${#MATRIX} returns length of array MATRIX.

# ==> $RANDOM%${#MATRIX} returns random number between 1
# ==> and length of MATRIX - 1.

# ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
# ==> returns expansion of MATRIX at random position, by length 1.
# ==> See {var:pos:len} parameter substitution in Section 3.3.1
# ==> and following examples.

# ==> PASS=... simply pastes this result onto previous PASS (concatenation).

# ==> To visualize this more clearly, uncomment the following line
# ==>             echo "$PASS"
# ==> to see PASS being built up,
# ==> one character at a time, each iteration of the loop.

let n+=1
# ==> Increment ’n’ for next pass.
done

echo "$PASS"      # ==> Or, redirect to file, as desired.

exit 0
```

+

James R. Van Zandt ha fornito questo script che fa uso delle named pipe ed è, parole sue, “vero esercizio di quoting ed escaping”.

Example A-16. fifo: Eseguire backup giornalieri utilizzando le pipe

```
#!/bin/bash
# ==> Script by James R. Van Zandt, and used here with his permission.

# ==> Comments added by author of this document.


  HERE=‘uname -n‘    # ==> hostname
  THERE=bilbo
  echo "starting remote backup to $THERE at ‘date +%r‘"
  # ==> ‘date +%r‘ returns time in 12-hour format, i.e. "08:08:34 PM".

  # make sure /pipe really is a pipe and not a plain file
  rm -rf /pipe
  mkfifo /pipe       # ==> Create a "named pipe", named "/pipe".

  # ==> ’su xyz’ runs commands as user "xyz".
  # ==> ’ssh’ invokes secure shell (remote login client).
  su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
  cd /
  tar -czf - bin boot dev etc home info lib man root sbin share usr var >/pipe
  # ==> Uses named pipe, /pipe, to communicate between processes:
  # ==> ’tar/gzip’ writes to /pipe and ’ssh’ reads from /pipe.

  # ==> The end result is this backs up the main directories, from / on down.

  # ==> What are the advantages of a "named pipe" in this situation,
  # ==> as opposed to an "anonymous pipe", with |?
  # ==> Will an anonymous pipe even work here?


  exit 0
```

+

Questo script, inviato da Stephane Chazelas, dimostra che si possono generare numeri primi senza ricorrere agli array.

Example A-17. Generare numeri primi utilizzando l’operatore modulo

```
#!/bin/bash
# primes.sh: Generate prime numbers, without using arrays.
# Script contributed by Stephane Chazelas.

#  This does *not* use the classic "Sieve of Eratosthenes" algorithm,
#+ but instead uses the more intuitive method of testing each candidate number
#+ for factors (divisors), using the "%" modulo operator.


LIMIT=1000                  # Primes 2 - 1000

Primes()
```

```
{
 (( n = $1 + 1 ))             # Bump to next integer.
 shift                        # Next parameter in list.
#  echo "_n=$n i=$i_"

 if (( n == LIMIT ))
 then echo $*
 return
 fi

 for i; do                    # "i" gets set to "@", previous values of $n.
#   echo "-n=$n i=$i-"
   (( i * i > n )) && break   # Optimization.
   (( n % i )) && continue    # Sift out non-primes using modulo operator.
   Primes $n $@               # Recursion inside loop.
   return
   done

   Primes $n $@ $n            # Recursion outside loop.
                              # Successively accumulate positional parameters.
                              # "$@" is the accumulating list of primes.
}

Primes 1

exit 0

#  Uncomment lines 16 and 24 to help figure out what is going on.

#  Compare the speed of this algorithm for generating primes
#+ with the Sieve of Eratosthenes (ex68.sh).

#  Exercise: Rewrite this script without recursion, for faster execution.

+
```

Jordi Sanfeliu ha dato il consenso all’uso del suo elegante script *tree*.

Example A-18. tree: Visualizzare l’albero di una directory

```
#!/bin/sh
#         @(#) tree      1.1  30/11/95       by Jordi Sanfeliu
#                                          email: mikaku@fiwix.org
#
#         Initial version:  1.0  30/11/95
#         Next version   :  1.1  24/02/97   Now, with symbolic links
#         Patch by       :  Ian Kjos, to support unsearchable dirs
#                           email: beth13@mail.utexas.edu
#
#         Tree is a tool for view the directory tree (obvious :-) )
#

# ==> ’Tree’ script used here with the permission of its author, Jordi Sanfeliu.
```

```
# ==> Comments added by the author of this document.
# ==> Argument quoting added.


search () {
   for dir in ‘echo *‘
   # ==> ‘echo *‘ lists all the files in current working directory, without line breaks.
   # ==> Similar effect to     for dir in *
   # ==> but "dir in ‘echo *‘" will not handle filenames with blanks.
   do
      if [ -d "$dir" ] ; then   # ==> If it is a directory (-d)...
         zz=0   # ==> Temp variable, keeping track of directory level.
         while [ $zz != $deep ]    # Keep track of inner nested loop.
         do
            echo -n "|   "    # ==> Display vertical connector symbol,
                      # ==> with 2 spaces & no line feed in order to indent.
            zz=‘expr $zz + 1‘ # ==> Increment zz.
         done
         if [ -L "$dir" ] ; then   # ==> If directory is a symbolic link...
            echo "+---$dir" ‘ls -l $dir | sed ’s/^.*’$dir’ //’‘
    # ==> Display horiz. connector and list directory name, but...
    # ==> delete date/time part of long listing.
         else
            echo "+---$dir"      # ==> Display horizontal connector symbol...
                                 # ==> and print directory name.
            if cd "$dir" ; then  # ==> If can move to subdirectory...
               deep=‘expr $deep + 1‘   # ==> Increment depth.
               search     # with recursivity ;-)
                  # ==> Function calls itself.
               numdirs=‘expr $numdirs + 1‘   # ==> Increment directory count.
            fi
         fi
      fi
   done
   cd ..   # ==> Up one directory level.
   if [ "$deep" ] ; then  # ==> If depth = 0 (returns TRUE)...
      swfi=1              # ==> set flag showing that search is done.
   fi
   deep=‘expr $deep - 1‘  # ==> Decrement depth.
}

# - Main -
if [ $# = 0 ] ; then
   cd ‘pwd‘    # ==> No args to script, then use current working directory.
else
   cd $1       # ==> Otherwise, move to indicated directory.
fi
echo "Initial directory = ‘pwd‘"
swfi=0      # ==> Search finished flag.
deep=0      # ==> Depth of listing.
numdirs=0
zz=0
```

```
while [ "$swfi" != 1 ]   # While flag not set...
do
   search   # ==> Call function after initializing variables.
done
echo "Total directories = $numdirs"

exit 0
# ==> Challenge: try to figure out exactly how this script works.
```

Noah Friedman ha permesso l'uso del suo script *string function*, che riproduce, sostanzialmente, alcune delle funzioni per la manipolazione di stringhe della libreria C.

Example A-19. string functions: Funzioni per stringhe simili a quelle del C

```
#!/bin/bash

# string.bash --- bash emulation of string(3) library routines
# Author: Noah Friedman <friedman@prep.ai.mit.edu>
# ==>     Used with his kind permission in this document.
# Created: 1992-07-01
# Last modified: 1993-09-29
# Public domain

# Conversion to bash v2 syntax done by Chet Ramey

# Commentary:
# Code:

#:docstring strcat:
# Usage: strcat s1 s2
#
# Strcat appends the value of variable s2 to variable s1.
#
# Example:
#    a="foo"
#    b="bar"
#    strcat a b
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload   ==> Autoloading of function commented out.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                        # indirect variable expansion
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}
```

```
#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Line strcat, but strncat appends a maximum of n characters from the value
# of variable s2.  It copies fewer if the value of variabl s2 is shorter
# than n characters.  Echoes result on stdout.
#
# Example:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}                      # ==> indirect variable expansion
    s2_val=${!s2}

    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}           # ==> substring extraction
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strcmp:
# Usage: strcmp $s1 $s2
#
# Strcmp compares its arguments and returns an integer less than, equal to,
# or greater than zero, depending on whether string s1 is lexicographically
# less than, equal to, or greater than string s2.
#:end docstring:

###;;;autoload
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
```

```
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Like strcmp, but makes the comparison by examining a maximum of n
# characters (n less than or equal to zero yields equality).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
```

```
    local result="${1%%[!${2}]*}"

    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr echoes a substring starting at the first occurrence of string s2 in
# string s1, or nothing if s2 does not occur in the string.  If s2 points to
# a string of zero length, strstr echoes s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # if s2 points to a string of zero length, strstr echoes s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # strstr echoes nothing if s2 does not occur in s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # use the pattern matching code to strip off the match and everything
    # following it
    first=${1/$2*/}

    # then strip off the first unmatched portion of the string
    echo "${1##$first}"
}

#:docstring strtok:
# Usage: strtok s1 s2
#
```

```
# Strtok considers the string s1 to consist of a sequence of zero or more
# text tokens separated by spans of one or more characters from the
# separator string s2.  The first call (with a non-empty string s1
# specified) echoes a string consisting of the first token on stdout. The
# function keeps track of its position in the string s1 between separate
# calls, so that subsequent calls made with the first argument an empty
# string will work through the string immediately following that token.  In
# this way subsequent calls will work through the string s1 until no tokens
# remain.  The separator string s2 may be different from call to call.
# When no token remains in s1, an empty value is echoed on stdout.
#:end docstring:

###;;;autoload
function strtok ()
{
 :
}

#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Used by many functions like strncmp to truncate arguments for comparison.
# Echoes the first n characters of each string s1 s2 ... on stdout.
#:end docstring:

###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# provide string

# string.bash ends here


# ========================================================================== #
# ==> Everything below here added by the document author.

# ==> Suggested use of this script is to delete everything below here,
# ==> and "source" this file into your own scripts.

# strcat
string0=one
string1=two
echo
echo "Testing \"strcat\" function:"
echo "Original \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
```

```
echo "New \"string0\" = $string0"
echo

# strlen
echo
echo "Testing \"strlen\" function:"
str=123456789
echo "\"str\" = $str"
echo -n "Length of \"str\" = "
strlen str
echo



# Exercise:
# --------
# Add code to test all the other string functions above.


exit 0
```

Esempio di array complesso, di Michael Zick, che utilizza il comando md5sum per codificare informazioni sulle directory.

Example A-20. Informazioni sulle directory

```
#! /bin/bash
# directory-info.sh
# Parses and lists directory information.

# NOTE: Change lines 273 and 353 per "README" file.

# Michael Zick is the author of this script.
# Used here with his permission.

# Controls
# If overridden by command arguments, they must be in the order:
#   Arg1: "Descriptor Directory"
#   Arg2: "Exclude Paths"
#   Arg3: "Exclude Directories"
#
# Environment Settings override Defaults.
# Command arguments override Environment Settings.

# Default location for content addressed file descriptors.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Directory paths never to list or enter
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Directories never to list or enter
declare -a \
```

```
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Files never to list or enter
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Name with Spaces")'}}


# Here document used as a comment block.
: << LSfieldsDoc
# # # # # List Filesystem Directory Information # # # # #
#
# ListDirectory "FileGlob" "Field-Array-Name"
# or
# ListDirectory -of "FileGlob" "Field-Array-Filename"
# '-of' meaning 'output to filename'
# # # # #

String format description based on: ls (GNU fileutils) version 4.0.36

Produces a line (or more) formatted:
inode permissions hard-links owner group ...
32736 -rw-------    1 mszick   mszick

size    day month date hh:mm:ss year path
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core

Unless it is formatted:
inode permissions hard-links owner group ...
266705 crw-rw----    1    root  uucp

major minor day month date hh:mm:ss year path
4,  68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: that pesky comma after the major number

NOTE: the 'path' may be multiple fields:
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

If that isn't enough to keep your parser guessing,
either or both of the path components may be relative:
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2

The first character of the 11 (10?) character permissions field:
's' Socket
'd' Directory
'b' Block device
'c' Character device
```

```
’l’ Symbolic link
NOTE: Hard links not marked - test for identical inode numbers
on identical filesystems.
All information about hard linked files are shared, except
for the names and the name’s location in the directory system.
NOTE: A "Hard link" is known as a "File Alias" on some systems.
’-’ An undistingushed file

Followed by three groups of letters for: User, Group, Others
Character 1: ’-’ Not readable; ’r’ Readable
Character 2: ’-’ Not writable; ’w’ Writable
Character 3, User and Group: Combined execute and special
’-’ Not Executable, Not Special
’x’ Executable, Not Special
’s’ Executable, Special
’S’ Not Executable, Special
Character 3, Others: Combined execute and sticky (tacky?)
’-’ Not Executable, Not Tacky
’x’ Executable, Not Tacky
’t’ Executable, Tacky
’T’ Not Executable, Tacky

Followed by an access indicator
Haven’t tested this one, it may be the eleventh character
or it may generate another field
’ ’ No alternate access
’+’ Alternate access
LSfieldsDoc


ListDirectory()
{
local -a T
local -i of=0 # Default return in variable
# OLD_IFS=$IFS # Using BASH default ’ \t\n’

case "$#" in
3) case "$1" in
-of) of=1 ; shift ;;
 * ) return 1 ;;
esac ;;
2) : ;; # Poor man’s "continue"
*) return 1 ;;
esac

# NOTE: the (ls) command is NOT quoted (")
T=( $(ls --inode --ignore-backups --almost-all --directory \
--full-time --color=none --time=status --sort=none \
--format=long $1) )

case $of in
# Assign T back to the array whose name was passed as $2
0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
```

```
# Write T into filename passed as $2
1) echo "${T[@]}" > "$2" ;;
esac
return 0
   }

# # # # # Is that string a legal number? # # # # #
#
# IsNumber "Var"
# # # # # There has to be a better way, sigh...

IsNumber()
{
local -i int
if [ $# -eq 0 ]
then
return 1
else
(let int=$1)  2>/dev/null
return $? # Exit status of the let thread
fi
}

# # # # # Index Filesystem Directory Information # # # # #
#
# IndexList "Field-Array-Name" "Index-Array-Name"
# or
# IndexList -if Field-Array-Filename Index-Array-Name
# IndexList -of Field-Array-Name Index-Array-Filename
# IndexList -if -of Field-Array-Filename Index-Array-Filename
# # # # #

: << IndexListDoc
Walk an array of directory fields produced by ListDirectory

Having suppressed the line breaks in an otherwise line oriented
report, build an index to the array element which starts each line.

Each line gets two index entries, the first element of each line
(inode) and the element that holds the pathname of the file.

The first index entry pair (Line-Number==0) are informational:
Index-Array-Name[0] : Number of "Lines" indexed
Index-Array-Name[1] : "Current Line" pointer into Index-Array-Name

The following index pairs (if any) hold element indexes into
the Field-Array-Name per:
Index-Array-Name[Line-Number * 2] : The "inode" field element.
NOTE: This distance may be either +11 or +12 elements.
Index-Array-Name[(Line-Number * 2) + 1] : The "pathname" element.
NOTE: This distance may be a variable number of elements.
Next line index pair for Line-Number+1.
IndexListDoc
```

```
IndexList()
{
local -a LIST # Local of listname passed
local -a -i INDEX=( 0 0 ) # Local of index to return
local -i Lidx Lcnt
local -i if=0 of=0 # Default to variable names

case "$#" in # Simplistic option testing
0) return 1 ;;
1) return 1 ;;
2) : ;; # Poor man’s continue
3) case "$1" in
-if) if=1 ;;
-of) of=1 ;;
 * ) return 1 ;;
   esac ; shift ;;
4) if=1 ; of=1 ; shift ; shift ;;
*) return 1
esac

# Make local copy of list
case "$if" in
0) eval LIST=\( \"\$\{$1\[@\]\}\" \) ;;
1) LIST=( $(cat $1) ) ;;
esac

# Grok (grope?) the array
Lcnt=${#LIST[@]}
Lidx=0
until (( Lidx >= Lcnt ))
do
if IsNumber ${LIST[$Lidx]}
then
local -i inode name
local ft
inode=Lidx
local m=${LIST[$Lidx+2]} # Hard Links field
ft=${LIST[$Lidx+1]:0:1}  # Fast-Stat
case $ft in
b) ((Lidx+=12)) ;; # Block device
c) ((Lidx+=12)) ;; # Character device
*) ((Lidx+=11)) ;; # Anything else
esac
name=Lidx
case $ft in
-) ((Lidx+=1)) ;; # The easy one
b) ((Lidx+=1)) ;; # Block device
c) ((Lidx+=1)) ;; # Character device
d) ((Lidx+=1)) ;; # The other easy one
l) ((Lidx+=3)) ;; # At LEAST two more fields
```

```
#  A little more elegance here would handle pipes,
#+ sockets, deleted files - later.
*) until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
do
((Lidx+=1))
done
;; # Not required
esac
INDEX[${#INDEX[*]}]=$inode
INDEX[${#INDEX[*]}]=$name
INDEX[0]=${INDEX[0]}+1 # One more "line" found
# echo "Line: ${INDEX[0]} Type: $ft Links: $m Inode: \
# ${LIST[$inode]} Name: ${LIST[$name]}"

else
((Lidx+=1))
fi
done
case "$of" in
0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
1) echo "${INDEX[@]}" > "$2" ;;
esac
return 0 # What could go wrong?
}

# # # # # Content Identify File # # # # # #
#
# DigestFile Input-Array-Name Digest-Array-Name
# or
# DigestFile -if Input-FileName Digest-Array-Name
# # # # #

# Here document used as a comment block.
: <<DigestFilesDoc

The key (no pun intended) to a Unified Content File System (UCFS)
is to distinguish the files in the system based on their content.
Distinguishing files by their name is just, so, 20th Century.

The content is distinguished by computing a checksum of that content.
This version uses the md5sum program to generate a 128 bit checksum
representative of the file's contents.
There is a chance that two files having different content might
generate the same checksum using md5sum (or any checksum).  Should
that become a problem, then the use of md5sum can be replace by a
cyrptographic signature.  But until then...

The md5sum program is documented as outputting three fields (and it
does), but when read it appears as two fields (array elements).  This
is caused by the lack of whitespace between the second and third field.
So this function gropes the md5sum output and returns:
[0] 32 character checksum in hexidecimal (UCFS filename)
[1] Single character: ' ' text file, '*' binary file
```

```
[2] Filesystem (20th Century Style) name
Note: That name may be the character ’-’ indicating STDIN read.

DigestFilesDoc



DigestFile()
{
local if=0 # Default, variable name
local -a T1 T2

case "$#" in
3) case "$1" in
-if) if=1 ; shift ;;
 * ) return 1 ;;
esac ;;
2) : ;; # Poor man’s "continue"
*) return 1 ;;
esac

case $if in
0) eval T1=\( \"\$\{$1\[@\]\}\" \)
   T2=( $(echo ${T1[@]} | md5sum -) )
   ;;
1) T2=( $(md5sum $1) )
   ;;
esac

case ${#T2[@]} in
0) return 1 ;;
1) return 1 ;;
2) case ${T2[1]:0:1} in # SanScrit-2.0.5
   \*) T2[${#T2[@]}]=${T2[1]:1}
       T2[1]=\*
       ;;
    *) T2[${#T2[@]}]=${T2[1]}
       T2[1]=" "
       ;;
   esac
   ;;
3) : ;; # Assume it worked
*) return 1 ;;
esac

local -i len=${#T2[0]}
if [ $len -ne 32 ] ; then return 1 ; fi
eval $2=\( \"\$\{T2\[@\]\}\" \)
}

# # # # # Locate File # # # # #
#
# LocateFile [-l] FileName Location-Array-Name
```

```
# or
# LocateFile [-l] -of FileName Location-Array-FileName
# # # # #

# A file location is Filesystem-id and inode-number

# Here document used as a comment block.
: <<StatFieldsDoc
Based on stat, version 2.2
stat -t and stat -lt fields
[0] name
[1] Total size
File - number of bytes
Symbolic link - string length of pathname
[2] Number of (512 byte) blocks allocated
[3] File type and Access rights (hex)
[4] User ID of owner
[5] Group ID of owner
[6] Device number
[7] Inode number
[8] Number of hard links
[9] Device type (if inode device) Major
[10] Device type (if inode device) Minor
[11] Time of last access
May be disabled in ’mount’ with noatime
atime of files changed by exec, read, pipe, utime, mknod (mmap?)
atime of directories changed by addition/deletion of files
[12] Time of last modification
mtime of files changed by write, truncate, utime, mknod
mtime of directories changed by addtition/deletion of files
[13] Time of last change
ctime reflects time of changed inode information (owner, group
permissions, link count
-*-*- Per:
Return code: 0
Size of array: 14
Contents of array
Element 0: /home/mszick
Element 1: 4096
Element 2: 8
Element 3: 41e8
Element 4: 500
Element 5: 500
Element 6: 303
Element 7: 32385
Element 8: 22
Element 9: 0
Element 10: 0
Element 11: 1051221030
Element 12: 1051214068
Element 13: 1051214068

For a link in the form of linkname -> realname
```

```
stat -t  linkname returns the linkname (link) information
stat -lt linkname returns the realname information

stat -tf and stat -ltf fields
[0] name
[1] ID-0? # Maybe someday, but Linux stat structure
[2] ID-0? # does not have either LABEL nor UUID
# fields, currently information must come
# from file-system specific utilities
These will be munged into:
[1] UUID if possible
[2] Volume Label if possible
Note: 'mount -l' does return the label and could return the UUID

[3] Maximum length of filenames
[4] Filesystem type
[5] Total blocks in the filesystem
[6] Free blocks
[7] Free blocks for non-root user(s)
[8] Block size of the filesystem
[9] Total inodes
[10] Free inodes

-*-*- Per:
Return code: 0
Size of array: 11
Contents of array
Element 0: /home/mszick
Element 1: 0
Element 2: 0
Element 3: 255
Element 4: ef53
Element 5: 2581445
Element 6: 2277180
Element 7: 2146050
Element 8: 4096
Element 9: 1311552
Element 10: 1276425

StatFieldsDoc


# LocateFile [-l] FileName Location-Array-Name
# LocateFile [-l] -of FileName Location-Array-FileName

LocateFile()
{
local -a LOC LOC1 LOC2
local lk="" of=0

case "$#" in
0) return 1 ;;
1) return 1 ;;
```

```
2) : ;;
*) while (( "$#" > 2 ))
   do
      case "$1" in
       -l) lk=-1 ;;
      -of) of=1 ;;
        *) return 1 ;;
      esac
   shift
           done ;;
esac

# More Sanscrit-2.0.5
      # LOC1=( $(stat -t $lk $1) )
      # LOC2=( $(stat -tf $lk $1) )
      # Uncomment above two lines if system has "stat" command installed.
LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
      ${LOC2[@]:1:2} ${LOC2[@]:4:1} )

case "$of" in
0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
1) echo "${LOC[@]}" > "$2" ;;
esac
return 0
# Which yields (if you are lucky, and have "stat" installed)
# -*-*- Location Discriptor -*-*-
# Return code: 0
# Size of array: 15
# Contents of array
# Element 0: /home/mszick 20th Century name
# Element 1: 41e8 Type and Permissions
# Element 2: 500 User
# Element 3: 500 Group
# Element 4: 303 Device
# Element 5: 32385 inode
# Element 6: 22 Link count
# Element 7: 0 Device Major
# Element 8: 0 Device Minor
# Element 9: 1051224608 Last Access
# Element 10: 1051214068 Last Modify
# Element 11: 1051214068 Last Status
# Element 12: 0 UUID (to be)
# Element 13: 0 Volume Label (to be)
# Element 14: ef53 Filesystem type
}



# And then there was some test code

ListArray() # ListArray Name
{
local -a Ta
```

```
eval Ta=\( \"\$\{$1\[@\]\}\" \)
echo
echo "-*-*- List of Array -*-*-"
echo "Size of array $1: ${#Ta[*]}"
echo "Contents of array $1:"
for (( i=0 ; i<${#Ta[*]} ; i++ ))
do
    echo -e "\tElement $i: ${Ta[$i]}"
done
return 0
}

declare -a CUR_DIR
# For small arrays
ListDirectory "${PWD}" CUR_DIR
ListArray CUR_DIR

declare -a DIR_DIG
DigestFile CUR_DIR DIR_DIG
echo "The new \"name\" (checksum) for ${CUR_DIR[9]} is ${DIR_DIG[0]}"

declare -a DIR_ENT
# BIG_DIR # For really big arrays - use a temporary file in ramdisk
# BIG-DIR # ListDirectory -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
ListDirectory "${CUR_DIR[11]}/*" DIR_ENT

declare -a DIR_IDX
# BIG-DIR # IndexList -if "/tmpfs/junk2" DIR_IDX
IndexList DIR_ENT DIR_IDX

declare -a IDX_DIG
# BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
# BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
DigestFile DIR_ENT IDX_DIG
# Small (should) be able to parallize IndexList & DigestFile
# Large (should) be able to parallize IndexList & DigestFile & the assignment
echo "The \"name\" (checksum) for the contents of ${PWD} is ${IDX_DIG[0]}"

declare -a FILE_LOC
LocateFile ${PWD} FILE_LOC
ListArray FILE_LOC

exit 0
```

Stephane Chazelas dà un esempio di programmazione object-oriented con uno script Bash.

Example A-21. Database object-oriented

```
#!/bin/bash
# obj-oriented.sh: Object-oriented programming in a shell script.
# Script by Stephane Chazelas.


person.new()        # Looks almost like a class declaration in C++.
{
  local obj_name=$1 name=$2 firstname=$3 birthdate=$4

  eval "$obj_name.set_name() {
          eval \"$obj_name.get_name() {
                   echo \$1
                 }\"
        }"

  eval "$obj_name.set_firstname() {
          eval \"$obj_name.get_firstname() {
                   echo \$1
                 }\"
        }"

  eval "$obj_name.set_birthdate() {
          eval \"$obj_name.get_birthdate() {
            echo \$1
          }\"
          eval \"$obj_name.show_birthdate() {
            echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")
          }\"
          eval \"$obj_name.get_age() {
            echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))
          }\"
        }"

  $obj_name.set_name $name
  $obj_name.set_firstname $firstname
  $obj_name.set_birthdate $birthdate
}

echo

person.new self Bozeman Bozo 101272413
# Create an instance of "person.new" (actually passing args to the function).

self.get_firstname       #   Bozo
self.get_name            #   Bozeman
self.get_age             #   28
self.get_birthdate       #   101272413
self.show_birthdate      #   Sat Mar 17 20:13:33 MST 1973

echo
```

```
# typeset -f
# to see the created functions (careful, it scrolls off the page).

exit 0
```

Ora uno script che fa qualcosa di veramente utile: installare e montare quelle graziose “chiavi” USB.

Example A-22. Montare le chiavi di memoria USB

```
#!/bin/bash
# ==> usb.sh
# ==> Script for mounting and installing pen/keychain USB storage devices.
# ==> Runs as root at system startup (see below).

#  This code is free software covered by GNU GPL license version 2 or above.
#  Please refer to http://www.gnu.org/ for the full license text.
#
#  Some code lifted from usb-mount by Michael Hamilton’s usb-mount (LGPL)
#+ see http://users.actrix.co.nz/michael/usbmount.html
#
#  INSTALL
#  -------
#  Put this in /etc/hotplug/usb/diskonkey.
#  Then look in /etc/hotplug/usb.distmap, and copy all usb-storage entries
#+ into /etc/hotplug/usb.usermap, substituting "usb-storage" for "diskonkey".
#  Otherwise this code is only run during the kernel module invocation/removal
#+ (at least in my tests), which defeats the purpose.
#
#  TODO
#  ----
#  Handle more than one diskonkey device at one time (e.g. /dev/diskonkey1
#+ and /mnt/diskonkey1), etc. The biggest problem here is the handling in
#+ devlabel, which I haven’t yet tried.
#
#  AUTHOR and SUPPORT
#  ------------------
#  Konstantin Riabitsev, <icon linux duke edu>.
#  Send any problem reports to my email address at the moment.
#
# ==> Comments added by ABS Guide author.



SYMLINKDEV=/dev/diskonkey
MOUNTPOINT=/mnt/diskonkey
DEVLABEL=/sbin/devlabel
DEVLABELCONFIG=/etc/sysconfig/devlabel
IAM=$0

##
# Functions lifted near-verbatim from usb-mount code.
#
function allAttachedScsiUsb {
```

```
    find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f | xargs grep -l 'Attached: Yes'
}
function scsiDevFromScsiUsb {
    echo $1 | awk -F"[-/]" '{ n=$(NF-1);  print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz", n+1,
 1) }'
}

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # lifted from usbcam code.
    #
    if [ -f /var/run/console.lock ]; then
        CONSOLEOWNER=`cat /var/run/console.lock`
    elif [ -f /var/lock/console.lock ]; then
        CONSOLEOWNER=`cat /var/lock/console.lock`
    else
        CONSOLEOWNER=
    fi
    for procEntry in $(allAttachedScsiUsb); do
        scsiDev=$(scsiDevFromScsiUsb $procEntry)
        #  Some bug with usb-storage?
        #  Partitions are not in /proc/partitions until they are accessed
        #+ somehow.
        /sbin/fdisk -l $scsiDev >/dev/null
        ##
        #  Most devices have partitioning info, so the data would be on
        #+ /dev/sd?1. However, some stupider ones don't have any partitioning
        #+ and use the entire device for data storage. This tries to
        #+ guess semi-intelligently if we have a /dev/sd?1 and if not, then
        #+ it uses the entire device and hopes for the better.
        #
        if grep -q `basename $scsiDev`1 /proc/partitions; then
            part="$scsiDev""1"
        else
            part=$scsiDev
        fi
        ##
        #  Change ownership of the partition to the console user so they can
        #+ mount it.
        #
        if [ ! -z "$CONSOLEOWNER" ]; then
            chown $CONSOLEOWNER:disk $part
        fi
        ##
        # This checks if we already have this UUID defined with devlabel.
        # If not, it then adds the device to the list.
        #
        prodid=`$DEVLABEL printid -d $part`
        if ! grep -q $prodid $DEVLABELCONFIG; then
            # cross our fingers and hope it works
            $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null
        fi
        ##
```

```
        # Check if the mount point exists and create if it doesn't.
        #
        if [ ! -e $MOUNTPOINT ]; then
            mkdir -p $MOUNTPOINT
        fi
        ##
        # Take care of /etc/fstab so mounting is easy.
        #
        if ! grep -q "^$SYMLINKDEV" /etc/fstab; then
            # Add an fstab entry
            echo -e \
                "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0" \
                >> /etc/fstab
        fi
    done
    if [ ! -z "$REMOVER" ]; then
        ##
        # Make sure this script is triggered on device removal.
        #
        mkdir -p `dirname $REMOVER`
        ln -s $IAM $REMOVER
    fi
elif [ "${ACTION}" = "remove" ]; then
    ##
    # If the device is mounted, unmount it cleanly.
    #
    if grep -q "$MOUNTPOINT" /etc/mtab; then
        # unmount cleanly
        umount -l $MOUNTPOINT
    fi
    ##
    # Remove it from /etc/fstab if it's there.
    #
    if grep -q "^$SYMLINKDEV" /etc/fstab; then
        grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
        mv -f /etc/.fstab.new /etc/fstab
    fi
fi
```

Come impedire alla shell di espandere e reinterpretare le stringhe?

Example A-23. Proteggere le stringhe letterali

```
#! /bin/bash
# protect_literal.sh

# set -vx

:<<-'_Protect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
```

```
    Revision: $ID$

    Documentation redirected to the Bash no-operation.
    Bash will '/dev/null' this block when the script is first read.
    (Uncomment the above set command to see this action.)

    Remove the first (Sha-Bang) line when sourcing this as a library
    procedure.  Also comment out the example use code in the two
    places where shown.


    Usage:
        _protect_literal_str 'Whatever string meets your ${fancy}'
        Just echos the argument to standard out, hard quotes
        restored.

        $(_protect_literal_str 'Whatever string meets your ${fancy}')
        as the right-hand-side of an assignment statement.

    Does:
        As the right-hand-side of an assignment, preserves the
        hard quotes protecting the contents of the literal during
        assignment.

    Notes:
        The strange names (_*) are used to avoid trampling on
        the user's chosen names when this is sourced as a
        library.

_Protect_Literal_String_Doc

# The 'for illustration' function form

_protect_literal_str() {

# Pick an un-used, non-printing character as local IFS.
# Not required, but shows that we are ignoring it.
    local IFS=$'\x1B'               # \ESC character

# Enclose the All-Elements-Of in hard quotes during assignment.
    local tmp=$'\x27'$@$'\x27'
#    local tmp=$'\"$@$'\"         # Even uglier.

    local len=${#tmp}               # Info only.
    echo $tmp is $len long.         # Output AND information.
}

# This is the short-named version.
_pls() {
    local IFS=$'x1B'                # \ESC character (not required)
    echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
}
```

```
# :<<-'_Protect_Literal_String_Test'
# # # Remove the above "# " to disable this code. # # #

# See how that looks when printed.
echo
echo "- - Test One - -"
_protect_literal_str 'Hello $user'
_protect_literal_str 'Hello "${username}"'
echo

# Which yields:
# - - Test One - -
# 'Hello $user' is 13 long.
# 'Hello "${username}"' is 21 long.

#  Looks as expected, but why all of the trouble?
#  The difference is hidden inside the Bash internal order
#+ of operations.
#  Which shows when you use it on the RHS of an assignment.

# Declare an array for test values.
declare -a arrayZ

# Assign elements with various types of quotes and escapes.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Now list that array and see what is there.
echo "- - Test Two - -"
for (( i=0 ; i<${#arrayZ[*]} ; i++ ))
do
    echo  Element $i: ${arrayZ[$i]} is: ${#arrayZ[$i]} long.
done
echo

# Which yields:
# - - Test Two - -
# Element 0: zero is: 4 long.           # Our marker element
# Element 1: 'Hello ${Me}' is: 13 long. # Our "$(_pls '...' )"
# Element 2: Hello ${You} is: 12 long.  # Quotes are missing
# Element 3: \'Pass: \' is: 10 long.    # ${pw} expanded to nothing

# Now make an assignment with that result.
declare -a array2=( ${arrayZ[@]} )

# And print what happened.
echo "- - Test Three - -"
for (( i=0 ; i<${#array2[*]} ; i++ ))
do
    echo  Element $i: ${array2[$i]} is: ${#array2[$i]} long.
done
echo

# Which yields:
```

```
# - - Test Three - -
# Element 0: zero is: 4 long.           # Our marker element.
# Element 1: Hello ${Me} is: 11 long.   # Intended result.
# Element 2: Hello is: 5 long.          # ${You} expanded to nothing.
# Element 3: 'Pass: is: 6 long.         # Split on the whitespace.
# Element 4: ' is: 1 long.              # The end quote is here now.

#  Our Element 1 has had its leading and trailing hard quotes stripped.
#  Although not shown, leading and trailing whitespace is also stripped.
#  Now that the string contents are set, Bash will always, internally,
#+ hard quote the contents as required during its operations.

#  Why?
#  Considering our "$(_pls 'Hello ${Me}')" construction:
#  " ... " -> Expansion required, strip the quotes.
#  $( ... ) -> Replace with the result of..., strip this.
#  _pls ' ... ' -> called with literal arguments, strip the quotes.
#  The result returned includes hard quotes; BUT the above processing
#+ has already been done, so they become part of the value assigned.
#
#  Similarly, during further usage of the string variable, the ${Me}
#+ is part of the contents (result) and survives any operations
#  (Until explicitly told to evaluate the string).

#  Hint: See what happens when the hard quotes ($'\x27') are replaced
#+ with soft quotes ($'\x22') in the above procedures.
#  Interesting also is to remove the addition of any quoting.

# _Protect_Literal_String_Test
# # # Remove the above "# " to disable this code. # # #

exit 0
```

E se si *volesse* che la shell espanda e reinterpreti le stringhe?

Example A-24. Stringhe letterali non protette

```
#! /bin/bash
# unprotect_literal.sh

# set -vx

:<<-'_UnProtect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Documentation redirected to the Bash no-operation. Bash will
    '/dev/null' this block when the script is first read.
    (Uncomment the above set command to see this action.)
```

```
    Remove the first (Sha-Bang) line when sourcing this as a library
    procedure.  Also comment out the example use code in the two
    places where shown.


    Usage:
        Complement of the "$(_pls 'Literal String')" function.
        (See the protect_literal.sh example.)

        StringVar=$(_upls ProtectedSringVariable)

    Does:
        When used on the right-hand-side of an assignment statement;
        makes the substitions embedded in the protected string.

    Notes:
        The strange names (_*) are used to avoid trampling on
        the user's chosen names when this is sourced as a
        library.


_UnProtect_Literal_String_Doc

_upls() {
    local IFS=$'x1B'               # \ESC character (not required)
    eval echo $@                   # Substitution on the glob.
}

# :<<-'_UnProtect_Literal_String_Test'
# # # Remove the above "# " to disable this code. # # #


_pls() {
    local IFS=$'x1B'               # \ESC character (not required)
    echo $'\x27'$@$'\x27'          # Hard quoted parameter glob
}

# Declare an array for test values.
declare -a arrayZ

# Assign elements with various types of quotes and escapes.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Now make an assignment with that result.
declare -a array2=( ${arrayZ[@]} )

# Which yielded:
# - - Test Three - -
# Element 0: zero is: 4 long            # Our marker element.
# Element 1: Hello ${Me} is: 11 long    # Intended result.
# Element 2: Hello is: 5 long           # ${You} expanded to nothing.
# Element 3: 'Pass: is: 6 long          # Split on the whitespace.
# Element 4: ' is: 1 long               # The end quote is here now.
```

```
# set -vx

#  Initialize ’Me’ to something for the embedded ${Me} substitution.
#  This needs to be done ONLY just prior to evaluating the
#+ protected string.
#  (This is why it was protected to begin with.)

Me="to the array guy."

# Set a string variable destination to the result.
newVar=$(_upls ${array2[1]})

# Show what the contents are.
echo $newVar

# Do we really need a function to do this?
newerVar=$(eval echo ${array2[1]})
echo $newerVar

#  I guess not, but the _upls function gives us a place to hang
#+ the documentation on.
#  This helps when we forget what a # construction like:
#+ $(eval echo ... ) means.

# What if Me isn’t set when the protected string is evaluated?
unset Me
newestVar=$(_upls ${array2[1]})
echo $newestVar

# Just gone, no hints, no runs, no errors.

#  Why in the world?
#  Setting the contents of a string variable containing character
#+ sequences that have a meaning to Bash is a general problem in
#+ script programming.
#
#  This problem is now solved in eight lines of code
#+ (and four pages of description).

#  Where is all this going?
#  Dynamic content Web pages as an array of Bash strings.
#  Content set per request by a Bash ’eval’ command
#+ on the stored page template.
#  Not intended to replace PHP, just an interesting thing to do.
###
#  Don’t have a webserver application?
#  No problem, check the example directory of the Bash source;
#+ there is a Bash script for that also.

# _UnProtect_Literal_String_Test
# # # Remove the above "# " to disable this code. # # #
```

```
exit 0
```

Per terminare la sezione, un ripasso dei fondamenti . . . ed altro.

Example A-25. Fondamenti rivisitati

```
#!/bin/bash
# basics-reviewed.bash

# File extension == *.bash == specific to Bash

#   Copyright (c) Michael S. Zick, 2003; All rights reserved.
#   License: Use in any form, for any purpose.
#   Revision: $ID$
#
#             Edited for layout by M.C.
#   (author of the "Advanced Bash Scripting Guide")


#  This script tested under Bash versions 2.04, 2.05a and 2.05b.
#  It may not work with earlier versions.
#  This demonstration script generates one --intentional--
#+ "command not found" error message. See line 394.

#  The current Bash maintainer, Chet Ramey, has fixed the items noted
#+ for an upcoming version of Bash.



        ###-------------------------------------------###
        ###  Pipe the output of this script to 'more' ###
        ###+ else it will scroll off the page.        ###
        ###                                           ###
        ###  You may also redirect its output         ###
        ###+ to a file for examination.               ###
        ###-------------------------------------------###



#  Most of the following points are described at length in
#+ the text of the foregoing "Advanced Bash Scripting Guide."
#  This demonstration script is mostly just a reorganized presentation.
#      -- msz

# Variables are not typed unless otherwise specified.

#  Variables are named. Names must contain a non-digit.
#  File descriptor names (as in, for example: 2>&1)
#+ contain ONLY digits.

# Parameters and Bash array elements are numbered.
# (Parameters are very similar to Bash arrays.)
```

```
# A variable name may be undefined (null reference).
unset VarNull

# A variable name may be defined but empty (null contents).
VarEmpty=”                          # Two, adjacent, single quotes.

# A variable name my be defined and non-empty
VarSomething=’Literal’

# A variable may contain:
#   * A whole number as a signed 32-bit (or larger) integer
#   * A string
# A variable may also be an array.

#  A string may contain embedded blanks and may be treated
#+ as if it where a function name with optional arguments.

#  The names of variables and the names of functions
#+ are in different namespaces.


#  A variable may be defined as a Bash array either explicitly or
#+ implicitly by the syntax of the assignment statement.
#  Explicit:
declare -a ArrayVar



# The echo command is a built-in.
echo $VarSomething

# The printf command is a built-in.
# Translate %s as: String-Format
printf %s $VarSomething         # No linebreak specified, none output.
echo                            # Default, only linebreak output.




# The Bash parser word breaks on whitespace.
# Whitespace, or the lack of it is significant.
# (This holds true in general; there are, of course, exceptions.)




# Translate the DOLLAR_SIGN character as: Content-Of.

# Extended-Syntax way of writing Content-Of:
echo ${VarSomething}

#  The ${ ... } Extended-Syntax allows more than just the variable
#+ name to be specified.
```

```
#  In general, $VarSomething can always be written as: ${VarSomething}.

# Call this script with arguments to see the following in action.



#  Outside of double-quotes, the special characters @ and *
#+ specify identical behavior.
#  May be pronounced as: All-Elements-Of.

#  Without specification of a name, they refer to the
#+ pre-defined parameter Bash-Array.



# Glob-Pattern references
echo $*                         # All parameters to script or function
echo ${*}                       # Same

# Bash disables filename expansion for Glob-Patterns.
# Only character matching is active.


# All-Elements-Of references
echo $@                         # Same as above
echo ${@}                       # Same as above




#  Within double-quotes, the behavior of Glob-Pattern references
#+ depends on the setting of IFS (Input Field Separator).
#  Within double-quotes, All-Elements-Of references behave the same.


#  Specifying only the name of a variable holding a string refers
#+ to all elements (characters) of a string.


#  To specify an element (character) of a string,
#+ the Extended-Syntax reference notation (see below) MAY be used.




#  Specifying only the name of a Bash array references
#+ the subscript zero element,
#+ NOT the FIRST DEFINED nor the FIRST WITH CONTENTS element.

#  Additional qualification is needed to reference other elements,
#+ which means that the reference MUST be written in Extended-Syntax.
#  The general form is: ${name[subscript]}.
```

```
#  The string forms may also be used: ${name:subscript}
#+ for Bash-Arrays when referencing the subscript zero element.


# Bash-Arrays are implemented internally as linked lists,
#+ not as a fixed area of storage as in some programming languages.


#   Characteristics of Bash arrays (Bash-Arrays):
#   ---------------------------------------------

#   If not otherwise specified, Bash-Array subscripts begin with
#+  subscript number zero. Literally: [0]
#   This is called zero-based indexing.
###
#   If not otherwise specified, Bash-Arrays are subscript packed
#+  (sequential subscripts without subscript gaps).
###
#   Negative subscripts are not allowed.
###
#   Elements of a Bash-Array need not all be of the same type.
###
#   Elements of a Bash-Array may be undefined (null reference).
#       That is, a Bash-Array my be "subscript sparse."
###
#   Elements of a Bash-Array may be defined and empty (null contents).
###
#   Elements of a Bash-Array may contain:
#     * A whole number as a signed 32-bit (or larger) integer
#     * A string
#     * A string formated so that it appears to be a function name
#     + with optional arguments
###
#   Defined elements of a Bash-Array may be undefined (unset).
#       That is, a subscript packed Bash-Array may be changed
#   +   into a subscript sparse Bash-Array.
###
#   Elements may be added to a Bash-Array by defining an element
#+  not previously defined.
###
# For these reasons, I have been calling them "Bash-Arrays".
# I'll return to the generic term "array" from now on.
#     -- msz




#  Demo time -- initialize the previously declared ArrayVar as a
#+ sparse array.
#  (The 'unset ... ' is just documentation here.)

unset ArrayVar[0]                  # Just for the record
ArrayVar[1]=one                    # Unquoted literal
```

```
ArrayVar[2]=”                       # Defined, and empty
unset ArrayVar[3]                    # Just for the record
ArrayVar[4]=’four’                   # Quoted literal



# Translate the %q format as: Quoted-Respecting-IFS-Rules.
echo
echo ’- - Outside of double-quotes - -’
###
printf %q ${ArrayVar[*]}            # Glob-Pattern All-Elements-Of
echo
echo ’echo command:’${ArrayVar[*]}
###
printf %q ${ArrayVar[@]}            # All-Elements-Of
echo
echo ’echo command:’${ArrayVar[@]}

# The use of double-quotes may be translated as: Enable-Substitution.

# There are five cases recognized for the IFS setting.

echo
echo ’- - Within double-quotes - Default IFS of space-tab-newline - -’
IFS=$’\x20’$’\x09’$’\x0A’           #  These three bytes,
                                    #+ in exactly this order.


printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo ’echo command:’"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo ’echo command:’"${ArrayVar[@]}"


echo
echo ’- - Within double-quotes - First character of IFS is ^ - -’
# Any printing, non-whitespace character should do the same.
IFS=’^’$IFS                         # ^ + space tab newline
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo ’echo command:’"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo ’echo command:’"${ArrayVar[@]}"


echo
echo ’- - Within double-quotes - Without whitespace in IFS - -’
```

```
IFS='^:%!'
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS set and empty - -'
IFS=''
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


echo
echo '- - Within double-quotes - IFS undefined - -'
unset IFS
###
printf %q "${ArrayVar[*]}"          # Glob-Pattern All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[*]}"
###
printf %q "${ArrayVar[@]}"          # All-Elements-Of
echo
echo 'echo command:'"${ArrayVar[@]}"


# Put IFS back to the default.
# Default is exactly these three bytes.
IFS=$'\x20'$'\x09'$'\x0A'           # In exactly this order.

# Interpretation of the above outputs:
#   A Glob-Pattern is I/O; the setting of IFS matters.
###
#   An All-Elements-Of does not consider IFS settings.
###
#   Note the different output using the echo command and the
#+  quoted format operator of the printf command.


#  Recall:
#   Parameters are similar to arrays and have the similar behaviors.
###
```

```
#  The above examples demonstrate the possible variations.
#  To retain the shape of a sparse array, additional script
#+ programming is required.
###
#  The source code of Bash has a routine to output the
#+ [subscript]=value   array assignment format.
#  As of version 2.05b, that routine is not used,
#+ but that might change in future releases.



# The length of a string, measured in non-null elements (characters):
echo
echo '- - Non-quoted references - -'
echo 'Non-Null character count: '${#VarSomething}' characters.'

# test='Lit'$'\x00"eral'           # $'\x00' is a null character.
# echo ${#test}                     # See that?



#  The length of an array, measured in defined elements,
#+ including null content elements.
echo
echo 'Defined content count: '${#ArrayVar[@]}' elements.'
# That is NOT the maximum subscript (4).
# That is NOT the range of the subscripts (1 . . 4 inclusive).
# It IS the length of the linked list.
###
#  Both the maximum subscript and the range of the subscripts may
#+ be found with additional script programming.

# The length of a string, measured in non-null elements (characters):
echo
echo '- - Quoted, Glob-Pattern references - -'
echo 'Non-Null character count: '"${#VarSomething}"' characters.'

#  The length of an array, measured in defined elements,
#+ including null-content elements.
echo
echo 'Defined element count: '"${#ArrayVar[*]}"' elements.'

#  Interpretation: Substitution does not effect the ${# ... } operation.
#  Suggestion:
#  Always use the All-Elements-Of character
#+ if that is what is intended (independence from IFS).



#  Define a simple function.
#  I include an underscore in the name
#+ to make it distinctive in the examples below.
###
```

```
#  Bash separates variable names and function names
#+ in different namespaces.
#  The Mark-One eyeball isn't that advanced.
###
_simple() {
    echo -n 'SimpleFunc'$@          #  Newlines are swallowed in
}                                   #+ result returned in any case.


# The ( ... ) notation invokes a command or function.
# The $( ... ) notation is pronounced: Result-Of.


# Invoke the function _simple
echo
echo '- - Output of function _simple - -'
_simple                             # Try passing arguments.
echo
# or
(_simple)                           # Try passing arguments.
echo

echo '- Is there a variable of that name? -'
echo $_simple not defined           # No variable by that name.

# Invoke the result of function _simple (Error msg intended)

###
$(_simple)                          # Gives an error message:
#                           line 394: SimpleFunc: command not found
#                           ---------------------------------------

echo
###

#  The first word of the result of function _simple
#+ is neither a valid Bash command nor the name of a defined function.
###
# This demonstrates that the output of _simple is subject to evaluation.
###
# Interpretation:
#   A function can be used to generate in-line Bash commands.


# A simple function where the first word of result IS a bash command:
###
_print() {
    echo -n 'printf %q '$@
}

echo '- - Outputs of function _print - -'
_print parm1 parm2                  # An Output NOT A Command.
echo
```

```
$(_print parm1 parm2)              #  Executes: printf %q parm1 parm2
                                   #  See above IFS examples for the
                                   #+ various possibilities.
echo

$(_print $VarSomething)            # The predictable result.
echo



# Function variables
# ------------------

echo
echo '- - Function variables - -'
# A variable may represent a signed integer, a string or an array.
# A string may be used like a function name with optional arguments.

# set -vx                          #  Enable if desired
declare -f funcVar                 #+ in namespace of functions

funcVar=_print                     # Contains name of function.
$funcVar parm1                     # Same as _print at this point.
echo

funcVar=$(_print )                 # Contains result of function.
$funcVar                           # No input, No output.
$funcVar $VarSomething             # The predictable result.
echo

funcVar=$(_print $VarSomething)    #  $VarSomething replaced HERE.
$funcVar                           #  The expansion is part of the
echo                               #+ variable contents.

funcVar="$(_print $VarSomething)"  #  $VarSomething replaced HERE.
$funcVar                           #  The expansion is part of the
echo                               #+ variable contents.

#  The difference between the unquoted and the double-quoted versions
#+ above can be seen in the "protect_literal.sh" example.
#  The first case above is processed as two, unquoted, Bash-Words.
#  The second case above is processed as one, quoted, Bash-Word.



# Delayed replacement
# -------------------

echo
echo '- - Delayed replacement - -'
funcVar="$(_print '$VarSomething')" # No replacement, single Bash-Word.
```

```
eval $funcVar                       # $VarSomething replaced HERE.
echo

VarSomething='NewThing'
eval $funcVar                       # $VarSomething replaced HERE.
echo

# Restore the original setting trashed above.
VarSomething=Literal

#  There are a pair of functions demonstrated in the
#+ "protect_literal.sh" and "unprotect_literal.sh" examples.
#  These are general purpose functions for delayed replacement literals
#+ containing variables.




# REVIEW:
# ------

#  A string can be considered a Classic-Array of elements (characters).
#  A string operation applies to all elements (characters) of the string
#+ (in concept, anyway).
###
#  The notation: ${array_name[@]} represents all elements of the
#+ Bash-Array: array_name.
###
#  The Extended-Syntax string operations can be applied to all
#+ elements of an array.
###
#  This may be thought of as a For-Each operation on a vector of strings.
###
#  Parameters are similar to an array.
#  The initialization of a parameter array for a script
#+ and a parameter array for a function only differ
#+ in the initialization of ${0}, which never changes its setting.
###
#  Subscript zero of the script's parameter array contains
#+ the name of the script.
###
#  Subscript zero of a function's parameter array DOES NOT contain
#+ the name of the function.
#  The name of the current function is accessed by the $FUNCNAME variable.
###
#  A quick, review list follows (quick, not short).

echo
echo '- - Test (but not change) - -'
echo '- null reference -'
echo -n ${VarNull-'NotSet'}' '          # NotSet
echo ${VarNull}                         # NewLine only
```

```
echo -n ${VarNull:-'NotSet'}' '         # NotSet
echo ${VarNull}                         # Newline only

echo '- null contents -'
echo -n ${VarEmpty-'Empty'}' '          # Only the space
echo ${VarEmpty}                        # Newline only
echo -n ${VarEmpty:-'Empty'}' '         # Empty
echo ${VarEmpty}                        # Newline only

echo '- contents -'
echo ${VarSomething-'Content'}          # Literal
echo ${VarSomething:-'Content'}         # Literal

echo '- Sparse Array -'
echo ${ArrayVar[@]-'not set'}

# ASCII-Art time
# State     Y==yes, N==no
#           -       :-
# Unset     Y       Y       ${# ... } == 0
# Empty     N       Y       ${# ... } == 0
# Contents  N       N       ${# ... } > 0

#  Either the first and/or the second part of the tests
#+ may be a command or a function invocation string.
echo
echo '- - Test 1 for undefined - -'
declare -i t
_decT() {
    t=$t-1
}

# Null reference, set: t == -1
t=${#VarNull}                           # Results in zero.
${VarNull- _decT }                      # Function executes, t now -1.
echo $t

# Null contents, set: t == 0
t=${#VarEmpty}                          # Results in zero.
${VarEmpty- _decT }                     # _decT function NOT executed.
echo $t

# Contents, set: t == number of non-null characters
VarSomething='_simple'                  # Set to valid function name.
t=${#VarSomething}                      # non-zero length
${VarSomething- _decT }                 # Function _simple executed.
echo $t                                 # Note the Append-To action.

# Exercise: clean up that example.
unset t
unset _decT
VarSomething=Literal
```

```
echo
echo '- - Test and Change - -'
echo '- Assignment if null reference -'
echo -n ${VarNull='NotSet'}' '          # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- Assignment if null reference -'
echo -n ${VarNull:='NotSet'}' '         # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- No assignment if null contents -'
echo -n ${VarEmpty='Empty'}' '          # Space only
echo ${VarEmpty}
VarEmpty=''

echo '- Assignment if null contents -'
echo -n ${VarEmpty:='Empty'}' '         # Empty Empty
echo ${VarEmpty}
VarEmpty=''

echo '- No change if already has contents -'
echo ${VarSomething='Content'}          # Literal
echo ${VarSomething:='Content'}         # Literal


# "Subscript sparse" Bash-Arrays
###
#  Bash-Arrays are subscript packed, beginning with
#+ subscript zero unless otherwise specified.
###
#  The initialization of ArrayVar was one way
#+ to "otherwise specify".  Here is the other way:
###
echo
declare -a ArraySparse
ArraySparse=( [1]=one [2]='' [4]='four' )
# [0]=null reference, [2]=null content, [3]=null reference

echo '- - Array-Sparse List - -'
# Within double-quotes, default IFS, Glob-Pattern

IFS=$'\x20'$'\x09'$'\x0A'
printf %q "${ArraySparse[*]}"
echo

#  Note that the output does not distinguish between "null content"
#+ and "null reference".
#  Both print as escaped whitespace.
###
#  Note also that the output does NOT contain escaped whitespace
#+ for the "null reference(s)" prior to the first defined element.
```

```
###
# This behavior of 2.04, 2.05a and 2.05b has been reported
#+ and may change in a future version of Bash.

#  To output a sparse array and maintain the [subscript]=value
#+ relationship without change requires a bit of programming.
#  One possible code fragment:
###
# local l=${#ArraySparse[@]}        # Count of defined elements
# local f=0                         # Count of found subscripts
# local i=0                         # Subscript to test
(                                   # Anonymous in-line function
    for (( l=${#ArraySparse[@]}, f = 0, i = 0 ; f < l ; i++ ))
    do
        # 'if defined then...'
        ${ArraySparse[$i]+ eval echo '\ ['$i']='${ArraySparse[$i]} ; (( f++ )) }
    done
)

# The reader coming upon the above code fragment cold
#+ might want to review "command lists" and "multiple commands on a line"
#+ in the text of the foregoing "Advanced Bash Scripting Guide."
###
#  Note:
#  The "read -a array_name" version of the "read" command
#+ begins filling array_name at subscript zero.
#  ArraySparse does not define a value at subscript zero.
###
#  The user needing to read/write a sparse array to either
#+ external storage or a communications socket must invent
#+ a read/write code pair suitable for their purpose.
###
# Exercise: clean it up.

unset ArraySparse

echo
echo '- - Conditional alternate (But not change)- -'
echo '- No alternate if null reference -'
echo -n ${VarNull+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- No alternate if null reference -'
echo -n ${VarNull:+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- Alternate if null contents -'
echo -n ${VarEmpty+'Empty'}' '              # Empty
echo ${VarEmpty}
VarEmpty="
```

```
echo ’- No alternate if null contents -’
echo -n ${VarEmpty:+’Empty’}’ ’             # Space only
echo ${VarEmpty}
VarEmpty=”

echo ’- Alternate if already has contents -’

# Alternate literal
echo -n ${VarSomething+’Content’}’ ’        # Content Literal
echo ${VarSomething}

# Invoke function
echo -n ${VarSomething:+ $(_simple) }’ ’    # SimpleFunc Literal
echo ${VarSomething}
echo

echo ’- - Sparse Array - -’
echo ${ArrayVar[@]+’Empty’}                 # An array of ’Empty’(ies)
echo

echo ’- - Test 2 for undefined - -’

declare -i t
_incT() {
    t=$t+1
}

#  Note:
#  This is the same test used in the sparse array
#+ listing code fragment.

# Null reference, set: t == -1
t=${#VarNull}-1                     # Results in minus-one.
${VarNull+ _incT }                  # Does not execute.
echo $t’ Null reference’

# Null contents, set: t == 0
t=${#VarEmpty}-1                    # Results in minus-one.
${VarEmpty+ _incT }                 # Executes.
echo $t’  Null content’

# Contents, set: t == (number of non-null characters)
t=${#VarSomething}-1                # non-null length minus-one
${VarSomething+ _incT }             # Executes.
echo $t’  Contents’

# Exercise: clean up that example.
unset t
unset _incT

# ${name?err_msg} ${name:?err_msg}
#  These follow the same rules but always exit afterwards
#+ if an action is specified following the question mark.
```

```
#  The action following the question mark may be a literal
#+ or a function result.
###
#  ${name?} ${name:?} are test-only, the return can be tested.



# Element operations
# ------------------

echo
echo '- - Trailing sub-element selection - -'

#  Strings, Arrays and Positional parameters

#  Call this script with multiple arguments
#+ to see the parameter selections.

echo '- All -'
echo ${VarSomething:0}              # all non-null characters
echo ${ArrayVar[@]:0}               # all elements with content
echo ${@:0}                         # all parameters with content;
                                    # ignoring parameter[0]

echo
echo '- All after -'
echo ${VarSomething:1}              # all non-null after character[0]
echo ${ArrayVar[@]:1}               # all after element[0] with content
echo ${@:2}                         # all after param[1] with content

echo
echo '- Range after -'
echo ${VarSomething:4:3}            # ral
                                    # Three characters after
                                    # character[3]

echo '- Sparse array gotch -'
echo ${ArrayVar[@]:1:2}     #  four - The only element with content.
                            #  Two elements after (if that many exist).
                            #  the FIRST WITH CONTENTS
                            #+ (the FIRST WITH  CONTENTS is being
                            #+ considered as if it
                            #+ were subscript zero).
#  Executed as if Bash considers ONLY array elements with CONTENT
#  printf %q "${ArrayVar[@]:0:3}"    # Try this one

#  In versions 2.04, 2.05a and 2.05b,
#+ Bash does not handle sparse arrays as expected using this notation.
#
#  The current Bash maintainer, Chet Ramey, has corrected this
#+ for an upcoming version of Bash.
```

```
echo '- Non-sparse array -'
echo ${@:2:2}               # Two parameters following parameter[1]

# New victims for string vector examples:
stringZ=abcABC123ABCabc
arrayZ=( abcabc ABCABC 123123 ABCABC abcabc )
sparseZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Victim string - -'$stringZ'- - '
echo ' - - Victim array - -'${arrayZ[@]}'- - '
echo ' - - Sparse array - -'${sparseZ[@]}'- - '
echo ' - [0]==null ref, [2]==null ref, [4]==null content - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - non-null-reference count: '${#sparseZ[@]}' elements'

echo
echo '- - Prefix sub-element removal - -'
echo '- - Glob-Pattern match must include the first character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
echo


# Function returning a simple, Literal, Glob-Pattern
_abc() {
    echo -n 'abc'
}

echo '- Shortest prefix -'
echo ${stringZ#123}                 # Unchanged (not a prefix).
echo ${stringZ#$(_abc)}             # ABC123ABCabc
echo ${arrayZ[@]#abc}               # Applied to each element.

# Fixed by Chet Ramey for an upcoming version of Bash.
# echo ${sparseZ[@]#abc}            # Version-2.05b core dumps.

# The -it would be nice- First-Subscript-Of
# echo ${#sparseZ[@]#*}             # This is NOT valid Bash.

echo
echo '- Longest prefix -'
echo ${stringZ##1*3}                # Unchanged (not a prefix)
echo ${stringZ##a*C}                # abc
echo ${arrayZ[@]##a*c}              # ABCABC 123123 ABCABC

# Fixed by Chet Ramey for an upcoming version of Bash
# echo ${sparseZ[@]##a*c}           # Version-2.05b core dumps.

echo
echo '- - Suffix sub-element removal - -'
echo '- - Glob-Pattern match must include the last character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
```

```
echo
echo '- Shortest suffix -'
echo ${stringZ%1*3}                # Unchanged (not a suffix).
echo ${stringZ%$(_abc)}            # abcABC123ABC
echo ${arrayZ[@]%abc}              # Applied to each element.

# Fixed by Chet Ramey for an upcoming version of Bash.
# echo ${sparseZ[@]%abc}           # Version-2.05b core dumps.

# The -it would be nice- Last-Subscript-Of
# echo ${#sparseZ[@]%*}            # This is NOT valid Bash.

echo
echo '- Longest suffix -'
echo ${stringZ%%1*3}               # Unchanged (not a suffix)
echo ${stringZ%%b*c}               # a
echo ${arrayZ[@]%%b*c}             # a ABCABC 123123 ABCABC a

# Fixed by Chet Ramey for an upcoming version of Bash.
# echo ${sparseZ[@]%%b*c}          # Version-2.05b core dumps.

echo
echo '- - Sub-element replacement - -'
echo '- - Sub-element at any location in string. - -'
echo '- - First specification is a Glob-Pattern - -'
echo '- - Glob-Pattern may be a literal or Glob-Pattern function result. - -'
echo '- - Second specification may be a literal or function result. - -'
echo '- - Second specification may be unspecified. Pronounce that'
echo '    as: Replace-With-Nothing (Delete) - -'
echo



# Function returning a simple, Literal, Glob-Pattern
_123() {
    echo -n '123'
}

echo '- Replace first occurrence -'
echo ${stringZ/$(_123)/999}        # Changed (123 is a component).
echo ${stringZ/ABC/xyz}            # xyzABC123ABCabc
echo ${arrayZ[@]/ABC/xyz}          # Applied to each element.
echo ${sparseZ[@]/ABC/xyz}         # Works as expected.

echo
echo '- Delete first occurrence -'
echo ${stringZ/$(_123)/}
echo ${stringZ/ABC/}
echo ${arrayZ[@]/ABC/}
echo ${sparseZ[@]/ABC/}

#  The replacement need not be a literal,
#+ since the result of a function invocation is allowed.
```

```
#  This is general to all forms of replacement.
echo
echo '- Replace first occurrence with Result-Of -'
echo ${stringZ/$(_123)/$(_simple)}  # Works as expected.
echo ${arrayZ[@]/ca/$(_simple)}     # Applied to each element.
echo ${sparseZ[@]/ca/$(_simple)}    # Works as expected.

echo
echo '- Replace all occurrences -'
echo ${stringZ//[b2]/X}             # X-out b's and 2's
echo ${stringZ//abc/xyz}            # xyzABC123ABCxyz
echo ${arrayZ[@]//abc/xyz}          # Applied to each element.
echo ${sparseZ[@]//abc/xyz}         # Works as expected.

echo
echo '- Delete all occurrences -'
echo ${stringZ//[b2]/}
echo ${stringZ//abc/}
echo ${arrayZ[@]//abc/}
echo ${sparseZ[@]//abc/}

echo
echo '- - Prefix sub-element replacement - -'
echo '- - Match must include the first character. - -'
echo

echo '- Replace prefix occurrences -'
echo ${stringZ/#[b2]/X}             # Unchanged (neither is a prefix).
echo ${stringZ/#$(_abc)/XYZ}        # XYZABC123ABCabc
echo ${arrayZ[@]/#abc/XYZ}          # Applied to each element.
echo ${sparseZ[@]/#abc/XYZ}         # Works as expected.

echo
echo '- Delete prefix occurrences -'
echo ${stringZ/#[b2]/}
echo ${stringZ/#$(_abc)/}
echo ${arrayZ[@]/#abc/}
echo ${sparseZ[@]/#abc/}

echo
echo '- - Suffix sub-element replacement - -'
echo '- - Match must include the last character. - -'
echo

echo '- Replace suffix occurrences -'
echo ${stringZ/%[b2]/X}             # Unchanged (neither is a suffix).
echo ${stringZ/%$(_abc)/XYZ}        # abcABC123ABCXYZ
echo ${arrayZ[@]/%abc/XYZ}          # Applied to each element.
echo ${sparseZ[@]/%abc/XYZ}         # Works as expected.

echo
echo '- Delete suffix occurrences -'
echo ${stringZ/%[b2]/}
```

```
echo ${stringZ/%$(_abc)/}
echo ${arrayZ[@]/%abc/}
echo ${sparseZ[@]/%abc/}

echo
echo '- - Special cases of null Glob-Pattern - -'
echo

echo '- Prefix all -'
# null substring pattern means 'prefix'
echo ${stringZ/#/NEW}               # NEWabcABC123ABCabc
echo ${arrayZ[@]/#/NEW}             # Applied to each element.
echo ${sparseZ[@]/#/NEW}            # Applied to null-content also.
                                    # That seems reasonable.

echo
echo '- Suffix all -'
# null substring pattern means 'suffix'
echo ${stringZ/%/NEW}               # abcABC123ABCabcNEW
echo ${arrayZ[@]/%/NEW}             # Applied to each element.
echo ${sparseZ[@]/%/NEW}            # Applied to null-content also.
                                    # That seems reasonable.

echo
echo '- - Special case For-Each Glob-Pattern - -'
echo '- - - - This is a nice-to-have dream - - - -'
echo

_GenFunc() {
    echo -n ${0}                    # Illustration only.
    # Actually, that would be an arbitrary computation.
}

# All occurrences, matching the AnyThing pattern.
# Currently //*/ does not match null-content nor null-reference.
# /#/ and /%/ does match null-content but not null-reference.
echo ${sparseZ[@]//*/$(_GenFunc)}


#  A possible syntax would be to make
#+ the parameter notation used within this construct mean:
#   ${1} - The full element
#   ${2} - The prefix, if any, to the matched sub-element
#   ${3} - The matched sub-element
#   ${4} - The suffix, if any, to the matched sub-element
#
# echo ${sparseZ[@]//*/$(_GenFunc ${3})}   # Same as ${1} here.
# Perhaps it will be implemented in a future version of Bash.


exit 0
```

Appendix B. Tabelle di riferimento

Le seguenti tabelle rappresentano un utile *riepilogo* di alcuni concetti dello scripting. Nella parte precedente del libro, questi argomenti sono trattati più approfonditamente, con esempi sul loro impiego.

Table B-1. Variabili speciali di shell

Variabile	Significato
`$0`	Nome dello script
`$1`	Parametro posizionale nr.1
`$2 - $9`	Parametri posizionali nr.2 - nr.9
`${10}`	Parametro posizionale nr.10
`$#`	Numero dei parametri posizionali
`"$*"`	Tutti i parametri posizionali (come parola singola) *
`"$@"`	Tutti i parametri posizionali (come stringhe separate)
`${#*}`	Numero dei parametri passati allo script da riga di comando
`${#@}`	Numero dei parametri passati allo script da riga di comando
`$?`	Valore di ritorno
`$$`	ID di processo (PID) dello script
`$-`	Opzioni passate allo script (usando *set*)
`$_`	Ultimo argomento del comando precedente
`$!`	ID di processo (PID) dell'ultimo job eseguito in background

***** *È necessario il quoting*, altrimenti viene trattato come "`$@`".

Table B-2. Operatori di verifica: confronti binari

Operatore	Significato	-----	Operatore	Significato
Confronto aritmetico			Confronto letterale	
`-eq`	Uguale a		`=`	Uguale a
			`==`	Uguale a
`-ne`	Non uguale a		`!=`	Non uguale a
`-lt`	Minore di		`\<`	Minore di (ASCII) *
`-le`	Minore di o uguale a			
`-gt`	Maggiore di		`\>`	Maggiore di (ASCII) *
`-ge`	Maggiore di o uguale a			
			`-z`	La stringa è vuota

Operatore	Significato	-----	Operatore	Significato
			`-n`	La stringa non è vuota
Confronto aritmetico	tra parentesi doppie ((...))			
`>`	Maggiore di			
`>=`	Maggiore di o uguale a			
`<`	Minore di			
`<=`	Minore di o uguale a			

***** *Se si usa il costrutto doppie parentesi quadre* [[...]] *, allora non è necessario il carattere di escape* \ .

Table B-3. Operatori di verifica: file

Operatore	Verifica se	-----	Operatore	Verifica se
`-e`	Il file esiste		`-s`	Il file non è vuoto
`-f`	Il file è un file *regolare*			
`-d`	Il file è una *directory*		`-r`	Il file ha il permesso di *lettura*
`-h`	Il file è un *link simbolico*		`-w`	Il file ha il permesso di *scrittura*
`-L`	Il file è un *link simbolico*		`-x`	Il file ha il permesso di *esecuzione*
`-b`	Il file è un *dispositivo a blocchi*			
`-c`	Il file è un *dispositivo a caratteri*		`-g`	È impostato il bit *sgid*
`-p`	Il file è una *pipe*		`-u`	È impostato il bit *suid*
`-S`	Il file è un *socket*		`-k`	È impostato lo “sticky bit”
`-t`	Il file è associato a un *terminale*			
`-N`	Il file è stato modificato dall’ultima lettura		`F1 -nt F2`	Il file F1 è *più recente* di F2 *
`-O`	Si è il proprietario del file		`F1 -ot F2`	Il file F1 è *più vecchio* di F2 *
`-G`	L’*id di gruppo* del file è uguale al vostro		`F1 -ef F2`	I file F1 e F2 sono degli *hard link* allo stesso file *

Operatore	Verifica se	-----	Operatore	Verifica se
!	"NOT" (inverte il senso delle precedenti verifiche)			

***** Operatore *binario* (richiede due operandi).

Table B-4. Sostituzione ed espansione di parametro

Espressione	Significato
`${var}`	Valore di *`var`*, uguale a *`$var`*
`${var-DEFAULT}`	Se *`var`* non è impostata, valuta l'espressione al valore di *`$DEFAULT`* *
`${var:-DEFAULT}`	Se *`var`* non è impostata o è vuota, valuta l'espressione al valore di *`$DEFAULT`* *
`${var=DEFAULT}`	Se *`var`* non è impostata, l'espressione è valutata al valore di *`$DEFAULT`* *
`${var:=DEFAULT}`	Se *`var`* non è impostata, l'espressione è valutata al valore di *`$DEFAULT`* *
`${var+ALTRO}`	Se *`var`* è impostata, l'espressione è valutata al valore di *`$ALTRO`*, altrimenti a stringa nulla
`${var:+ALTRO}`	Se *`var`* è impostata, l'espressione è valutata al valore di *`$ALTRO`*, altrimenti a stringa nulla
`${var?MSG_ERR}`	Se *`var`* non è impostata, visualizza *`$MSG_ERR`* *
`${var:?MSG_ERR}`	Se *`var`* non è impostata, visualizza *`$MSG_ERR`* *
`${!varprefix*}`	Verifica tutte le variabili dichiarate precedentemente i cui nomi iniziano con *`varprefix`*
`${!varprefix@}`	Verifica tutte le variabili dichiarate precedentemente i cui nomi iniziano con *`varprefix`*

***** Naturalmente se *`var`* *è* impostata, l'espressione viene valutata al valore di *`$var`*.

Table B-5. Operazioni su stringhe

Espressione	Significato
`${#stringa}`	Lunghezza di *`$stringa`*
`${stringa:posizione}`	Estrae la sottostringa da *`$stringa`* iniziando da *`$posizione`*

Espressione	**Significato**
`${stringa:posizione:lunghezza}`	Estrae una sottostringa di *`$lunghezza`* caratteri da *`$stringa`* iniziando da *`$posizione`*
`${stringa#sottostringa}`	Toglie l'occorrenza più breve di *`$sottostringa`* dalla parte iniziale di *`$stringa`*
`${stringa##sottostringa}`	Toglie l'occorrenza più lunga di *`$sottostringa`* dalla parte iniziale di *`$stringa`*
`${stringa%sottostringa}`	Toglie l'occorrenza più breve di *`$sottostringa`* dalla parte finale di *`$stringa`*
`${stringa%%sottostringa}`	Toglie l'occorrenza più lunga di *`$sottostringa`* dalla parte finale di *`$stringa`*
`${stringa/sottostringa/sostituto}`	Sostituisce la prima occorrenza di *`$sottostringa`* con *`$sostituto`*
`${stringa//sottostringa/sostituto}`	Sostituisce *tutte* le occorrenze di *`$sottostringa`* con *`$sostituto`*
`${stringa/#sottostringa/sostituto}`	Se *`$sottostringa`* viene verificata nella parte *iniziale* di *`$stringa`*, allora *`$sottostringa`* viene sostituita con *`$sostituto`*
`${stringa/%sottostringa/sostituto}`	Se *`$sottostringa`* viene verificata nella parte *finale* di *`$stringa`*, allora *`$sottostringa`* viene sostituita con *`$sostituto`*
`expr match "$stringa" '$sottostringa'`	Lunghezza di *`$sottostringa`** verificata nella parte iniziale di *`$stringa`*
`expr "$stringa" : '$sottostringa'`	Lunghezza di *`$sottostringa`** verificata nella parte iniziale di *`$stringa`*
`expr index "$stringa" $sottostringa`	Posizione numerica in *`$stringa`* del primo carattere compreso in *`$sottostringa`* che è stato verificato
`expr substr $stringa $posizione $lunghezza`	Estrae *`$lunghezza`* caratteri da *`$stringa`* iniziando da *`$posizione`*
`expr match "$stringa" '\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte iniziale di *`$stringa`*
`expr "$stringa" : '\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte iniziale di *`$stringa`*
`expr match "$stringa" '.*\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte finale di *`$stringa`*
`expr "$stringa" : '.*\($sottostringa\)'`	Estrae *`$sottostringa`** dalla parte finale di *`$stringa`*

***** Dove *`$sottostringa`* è un'*espressione regolare*.

Table B-6. Costrutti vari

Espressione	Interpretazione
Parentesi quadre	
`if [ CONDIZIONE ]`	Costrutto di verifica
`if [[ CONDIZIONE ]]`	Costrutto di verifica esteso
`Array[1]=elemento1`	Inizializzazione di array
`[a-z]`	Intervallo di caratteri in un'Espressione Regolare
Parentesi graffe	
`${variabile}`	Sostituzione di parametro
`${!variabile}`	Referenziazione indiretta di variabile
`{ comando1; comando2 }`	Blocco di codice
`{stringa1,stringa2,stringa3,...}`	Espansione multipla
Parentesi	
`( comando1; comando2 )`	Gruppo di comandi eseguiti in una subshell
`Array=(elemento1 elemento2 elemento3)`	Inizializzazione di array
`risultato=$(COMANDO)`	Esegue il comando in una subshell e assegna il risultato alla variabile
`>(COMANDO)`	Sostituzione di processo
`<(COMANDO)`	Sostituzione di processo
Doppie parentesi	
`(( var = 78 ))`	Aritmetica di interi
`var=$(( 20 + 5 ))`	Aritmetica di interi con assegnamento di variabile
Quoting	
`"$variabile"`	Quoting "debole"
`'stringa'`	Quoting "forte"
Apici inversi	
`risultato=‘COMANDO‘`	Esegue il comando in una subshell e assegna il risultato alla variabile

Appendix C. Una breve introduzione a Sed e Awk

Questa è una brevissima introduzione alle utility di elaborazione di testo **sed** e **awk**. Qui verranno trattati solamente alcuni comandi di base, ma che saranno sufficienti per comprendere i semplici costrutti sed e awk presenti negli script di shell.

sed: editor non interattivo di file di testo

awk: linguaggio per l'elaborazione di modelli orientato ai campi, con una sintassi simile a quella del C

Nonostante le loro differenze, le due utility condividono una sintassi d'invocazione simile, entrambe fanno uso delle espressioni regolari, entrambe leggono l'input, in modo predefinito, dallo `stdin` ed entrambe inviano i risultati allo `stdout`. Sono strumenti UNIX ben collaudati e funzionano bene insieme. L'output dell'una può essere collegato, per mezzo di una pipe, all'altra e le loro capacità combinate danno agli script di shell parte della potenza di Perl.

> **Note:** Un'importante differenza tra le due utility è che, mentre gli script di shell possono passare facilmente degli argomenti a sed, è più complicato fare la stessa cosa con awk (vedi Example 34-3 e Example 9-22).

C.1. Sed

Sed è un editor di riga non interattivo. Riceve un testo come input, o dallo `stdin` o da un file, esegue alcune operazioni sulle righe specificate, una alla volta, quindi invia il risultato allo `stdout` o in un file. Negli script di shell, sed è, di solito, uno delle molte componenti di una pipe.

Sed determina le righe dell'input, su cui deve operare, tramite un *indirizzo* che gli è stato passato. [1] Questo indirizzo può essere rappresentato sia da un numero di riga sia da una verifica d'occorrenza. Ad esempio, *`3d`* indica a sed di cancellare la terza riga dell'input, mentre *`/windows/d`* segnala a sed che si vogliono cancellare tutte le righe dell'input contenenti l'occorrenza "windows".

Di tutte le operazioni a disposizione di sed, vengono focalizzate, in primo luogo, le tre più comunemente usate. Esse sono **p**rint (visualizza allo `stdout`), **d**elete (cancella) e **s**ubstitute (sostituisce).

Table C-1. Operatori sed di base

Operatore	Nome	Effetto
`[indirizzo]/p`	print	Visualizza [l'indirizzo specificato]
`[indirizzo]/d`	delete	Cancella [l'indirizzo specificato]
`s/modello1/modello2/`	substitute	Sostituisce in ogni riga la prima occorrenza della stringa modello1 con la stringa modello2
`[indirizzo]/s/modello1/modello2/`	substitute	Sostituisce, in tutte le righe specificate in *`indirizzo`*, la prima occorrenza della stringa modello1 con la stringa modello2

Operatore	Nome	Effetto
`[indirizzo]/y/modello1/modello2/`	transform	sostituisce tutti i caratteri della stringa modello1 con i corrispondenti caratteri della stringa modello2, in tutte le righe specificate da *`indirizzo`* (equivalente di **tr**)
`g`	global	Agisce su *tutte* le verifiche d'occorrenza di ogni riga di input controllata

> **Note:** Se l'operatore `g` (*global*) non è accodato al comando substitute, la sostituzione agisce solo sulla prima verifica d'occorrenza di ogni riga.

Sia da riga di comando che in uno script di shell, un'operazione sed può richiedere il quoting e alcune opzioni.

```
sed -e '/^$/d' $nomefile
#  L'opzione -e indica che la stringa successiva deve essere interpretata come
#+ un'istruzione di editing.
#  (se a "sed" viene passata un'unica istruzione, "-e" è facoltativo.)
#  Il quoting "forte" (") protegge i caratteri delle ER presenti
#+ nell'istruzione dalla reinterpretazione, come caratteri speciali,
#+ da parte dello script.
# (Questo riserva solo a sed l'espansione delle ER.)
#
# Agisce sul testo del file $nomefile.
```

In certi casi, un comando di editing **sed** non funziona in presenza degli apici singoli.

```
nomefile=file1.txt
modello=INIZIO

  sed "/^$modello/d" "$nomefile"  # Funziona come indicato.
# sed '/^$modello/d' "$nomefile"    dà risultati imprevisti.
#        In questo esempio, il quoting forte (' ... '),
#+       impedisce l'espansione a "INIZIO" di "$modello".
```

> **Note:** Sed utilizza l'opzione `-e` per indicare che la stringa che segue è un'istruzione, o una serie di istruzioni. Se la stringa contiene una singola istruzione, allora questa opzione può essere omessa.

```
sed -n '/xzy/p' $nomefile
#  L'opzione -n indica a sed di visualizzare solo quelle righe che verificano
#+ il modello.
#  Altrimenti verrebbero visualizzate tutte le righe dell'input.
```

```
#  L’opzione -e, in questo caso, non sarebbe necessaria perché vi è una sola
#+ istruzione di editing.
```

Table C-2. Esempi di operatori sed

Notazione	Effetto
`8d`	Cancella l’ottava riga dell’input.
`/^$/d`	Cancella tutte le righe vuote.
`1,/^$/d`	Cancella dall’inizio dell’input fino alla prima riga vuota compresa.
`/Jones/p`	Visualizza solo le righe in cui è presente “Jones” (con l’opzione -n).
`s/Windows/Linux/`	Sostituisce con “Linux” la prima occorrenza di “Windows” trovata in ogni riga dell’input.
`s/BSOD/stabilità/g`	Sostituisce con “stabilità” tutte le occorrenze di “BSOD” trovate in ogni riga dell’input.
`s/ *$//`	Cancella tutti gli spazi che si trovano alla fine di ogni riga.
`s/00*/0/g`	Riduce ogni sequenza consecutiva di zeri ad un unico zero.
`/GUI/d`	Cancella tutte le righe in cui è presente “GUI”.
`s/GUI//g`	Cancella tutte le occorrenze di “GUI”, lasciando inalterata la parte restante di ciascuna riga.

Sostituire una stringa con un’altra di lunghezza zero (nulla) equivale a cancellare quella stringa nella riga di input. Questo lascia intatta la parte restante della riga. L’espressione **`s/GUI//`** applicata alla riga

`Le parti più importanti di ogni applicazione sono le sue GUI e gli effetti sonori`

dà come risultato

```
Le parti più importanti di ogni applicazione sono le sue e gli effetti sonori
```

La barra inversa rappresenta, come carattere di sostituzione, un *a capo*. In questo caso particolare, l’espressione di sostituzione deve continuare nella riga successiva.

```
s/^  */\
/g
```

Tutti gli spazi che si trovano all’inizio della riga vengono sostituiti con un carattere di a capo. Il risultato finale è la sostituzione di tutte le indentazioni dei paragrafi con righe vuote poste tra gli stessi paragrafi.

Un indirizzo seguito da una, o più, operazioni può richiedere l’impiego della parentesi graffa aperta e chiusa, con un uso appropriato dei caratteri di a capo.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
```

```
}
```

Questo cancella solo la prima di ogni serie di righe vuote. Potrebbe essere utile per effettuare la spaziatura singola di un file di testo mantenendo, però, la/e riga/he vuota/e tra i paragrafi.

> **Tip:** La via più rapida per effettuare una spaziatura doppia di un file di testo è **`sed G nomefile`**.

Per esempi che illustrano l'uso di sed negli script di shell, vedi:

1. Example 34-1
2. Example 34-2
3. Example 12-2
4. Example A-3
5. Example 12-13
6. Example 12-21
7. Example A-13
8. Example A-18
9. Example 12-25
10. Example 10-9
11. Example 12-34
12. Example A-2
13. Example 12-11
14. Example 12-9
15. Example A-11
16. Example 17-12

Per una spiegazione più ampia di sed, si controllino i relativi riferimenti in *Bibliography*.

C.2. Awk

Awk è un linguaggio completo per l'elaborazione di testo, con una sintassi che ricorda quella del **C**. Sebbene possegga un'ampia serie di funzionalità e di operatori, qui ne verranno analizzati solo un paio - quelli più utili allo scripting di shell.

Awk suddivide ogni riga dell'input che gli è stato passato in *campi*. Normalmente, un campo è una stringa di caratteri consecutivi separati da spazi, anche se esistono opzioni per modificare il delimitatore. Awk, quindi, analizza e agisce su ciascun singolo campo. Questo lo rende ideale per trattare file di testo strutturati -- in particolare le tabelle -- e dati organizzati in spezzoni logici, come righe e colonne.

Negli script di shell, i segmenti di codice awk vengono racchiusi da apici singoli (quoting forte) e da parentesi graffe.

```
awk ’{print $3}’ $nomefile
# Visualizza allo stdout il campo nr.3 del file $nomefile.

awk ’{print $1 $5 $6}’ $nomefile
# Visualizza i campi nr.1, 5 e 6 del file $nomefile.
```

Si è appena visto il comando **print** di awk in azione. L’altra sola funzionalità di awk di cui è necessaria la spiegazione sono le variabili. Awk le tratta in modo simile a come sono gestite negli script di shell, anche se con una maggiore flessibilità.

```
{ totale += ${numero_colonna} }
```

In questo modo si aggiunge il valore di *numero_colonna* al totale di “totale”. Infine, per visualizzare “totale”, vi è il comando di blocco di codice **END**, da eseguire dopo che lo script ha elaborato completamente il proprio input.

```
END { print totale }
```

Corrispondente ad **END**, vi è **BEGIN**, per il blocco di codice che deve essere eseguito prima che awk inizi l’elaborazione del suo input.

L’esempio seguente illustra come **awk** permetta di incrementare il numero di strumenti di veridica di testo a disposizione dello scripting di shell.

Example C-1. Conteggio delle occorrenze di lettera

```
#! /bin/sh
# letter-count.sh: Conta le occorrenze di lettere in un file di testo.
#
# Script di nyal (nyal@voila.fr).
# Usato con il permesso dell’autore.
# Ricommentato dall’autore di questo libro.


INIT_TAB_AWK=""
# Parametro per inizializzare lo script awk.
conteggio=0
FILE_INDICATO=$1

E_ERR_PARAM=65

utilizzo ()
{
    echo "Utilizzo: letter-count.sh file lettere" 2>&1
    # Per esempio:   ./letter-count.sh nomefile.txt a b c
    exit $E_ERR_PARAM # Parametri passati allo script insufficienti.
}
```

```
if [ ! -f "$1" ] ; then
    echo "$1: File inesistente." 2>&1
    utilizzo              # Visualizza il messaggio di utilizzo ed esce.
fi

if [ -z "$2" ] ; then
    echo "$2: Non è stata specificata nessuna lettera." 2>&1
    utilizzo
fi

shift                     # Le lettere sono state specificate.
for lettera in ‘echo $@‘   # Per ognuna . . .
  do
  INIT_TAB_AWK="$INIT_TAB_AWK tab_search[${conteggio}] = \"$lettera\";\
   final_tab[${conteggio}] = 0; "
  # Passato come parametro al successivo script awk.
  conteggio=‘expr $conteggio + 1‘
done

# DEBUGGING:
# echo $INIT_TAB_AWK;

cat $FILE_INDICATO |
# Il file viene collegato, per mezzo di una pipe, al seguente script awk.

# -------------------------------------------------------------------------
awk -v tab_search=0 -v final_tab=0 -v tab=0 -v nb_letter=0 -v chara=0 -v chara2=0 \
"BEGIN { $INIT_TAB_AWK } \
{ split(\$0, tab, \"\"); \
for (chara in tab) \
{ for (chara2 in tab_search) \
{ if (tab_search[chara2] == tab[chara]) { final_tab[chara2]++ } } } } \
END { for (chara in final_tab) \
{ print tab_search[chara] \" => \" final_tab[chara] } }"
# -------------------------------------------------------------------------
#  Niente di così complicato, solo . . .
#+ cicli for, costrutti if e un paio di funzioni specializzate.

exit $?
```

Per dimostrazioni più semplici dell’uso di awk negli script di shell, vedi:

1. Example 11-11
2. Example 16-7
3. Example 12-25
4. Example 34-3
5. Example 9-22
6. Example 11-17

7. Example 28-1
8. Example 28-2
9. Example 10-3
10. Example 12-43
11. Example 9-27
12. Example 12-3
13. Example 9-12
14. Example 34-12
15. Example 10-8

Questo è tutto, per quanto riguarda awk, ma vi sono moltissime altre cose da imparare. Si vedano i relativi riferimenti in *Bibliography*.

Notes

1. Se non viene specificato alcun indirizzo, sed, in modo predefinito, considera *tutte* le righe.

Appendix D. Codici di Exit con significati speciali

Table D-1. Codici di Exit “riservati”

Numero di codice Exit	Significato	Esempio	Commenti
`1`	indica errori generici	let "var1 = 1/0"	errori vari, come “divisione per zero”
`2`	uso scorretto dei builtin di shell, secondo la documentazione Bash		Si vede raramente, sostituito solitamente dal codice di exit 1
`126`	il comando invocato non può essere eseguito		problemi di permessi o il comando non è un eseguibile
`127`	“command not found”		possibili problemi con `$PATH` o errore di digitazione
`128`	argomento di exit non valido	exit 3.14159	**exit** richiede come argomento solo un intero compreso nell’intervallo 0 - 255
`128+n`	segnale di errore fatale “n”	**kill -9** `$PPID` dello script	**$?** restituisce 137 (128 + 9)
`130`	script terminato con Control-C		Control-C invia il segnale di errore fatale 2 (130 = 128 + 2, vedi sopra)
`255*`	exit status fuori intervallo	exit -1	**exit** richiede come argomento solo un intero compreso nell’intervallo 0 - 255

Secondo la tabella, i codici di exit 1 - 2, 126 - 165 e 255 [1] hanno significati speciali e, quindi, si dovrebbe evitare di usarli come parametri di exit definiti dall’utente. Terminare uno script con **exit 127** sicuramente provoca della confusione nella fase di risoluzione dei problemi (si tratta dell’errore “command not found” oppure è quello definito dall’utente?). Comunque, molti script usano **exit 1** come codice di uscita di errore generico. Dal momento che il codice di exit 1 può indicare molti differenti errori, questo, se da una parte non aggiunge ulteriore ambiguità, dall’altra non è neanche molto significativo.

Vi è stato un tentativo per sistematizzare i numeri degli exit status (vedi `/usr/include/sysexits.h`), ma questo fu fatto solo per i programmatori di C e C++. Uno standard simile sarebbe stato appropriato anche per lo scripting. L’autore di questo documento propone di limitare i codici di exit definiti dall’utente all’intervallo 64 - 113 (in aggiunta a 0, per indicare successo), per conformarsi allo standard del C/C++. In questo modo, si possono assegnare 50 validi codici rendendo, di fatto, più immediata la soluzione dei problemi degli script.

Tutti i codici di exit definiti dall’utente presenti negli esempi che accompagnano questo documento sono conformi a questo standard, tranne nei casi in cui vi siano state delle circostanze tali da non permettere l’applicazione di questa

regola, come in Example 9-2.

> **Note:** Eseguendo, dopo il termine di uno script, $? da riga di comando, si otterranno risultati coerenti con la precedente tabella, solo con Bash o *sh*. L'esecuzione di C-shell o *tcsh* potrebbe, in alcuni casi, fornire valori diversi.

Notes

1. Valori di exit al di fuori dell'intervallo possono dar luogo a numeri di codice imprevedibili. Per esempio, **exit 3809** dà come codice di exit 225.

Appendix E. Una dettagliata introduzione all'I/O e alla redirezione I/O

scritta da Stephane Chazelas e rivista dall'autore del documento

Un comando si aspetta che siano disponibili i primi tre descrittori di file. Il primo, *fd 0* (lo standard input, `stdin`), è utilizzato per la lettura. Gli altri due (*fd 1*, `stdout` e *fd 2*, `stderr`) per la scrittura.

Ad ogni comando sono associati uno `stdin`, uno `stdout` e uno `stderr`. **`ls 2>&1`** trasforma temporaneamente lo `stderr` del comando **ls** in un'unica "risorsa", lo `stdout` della shell.

Per convenzione, un comando legge il proprio input da fd 0 (`stdin`), visualizza l'output in fd 1 (`stdout`) e i messaggi d'errore in fd 2 (`stderr`). Se uno di questi descrittori di file non è aperto, si possono riscontrare dei problemi:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

Ad esempio, quando viene posto in esecuzione **xterm**, come prima cosa questo inizializza se stesso. Prima di mettere in esecuzione la shell dell'utente, **xterm** apre per tre volte il dispositivo di terminale (/dev/pts/<n> o qualcosa di analogo).

A questo punto Bash eredita questi tre descrittori di file, a loro volta ereditati da ogni comando (processo figlio) messo in esecuzione da Bash, tranne nel caso in cui il comando viene rediretto. Redirezione vuol dire riassegnare uno dei descrittori di file a un altro file (o ad una pipe, o ad altro che lo consenta). I descrittori di file possono essere riassegnati localmente (per un comando, un gruppo di comandi, una subshell, if o case, cicli for o while...), oppure globalmente, per l'intera shell (usando exec).

`ls > /dev/null` esegue **ls** con il suo fd 1 connesso a `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1
```

```
bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

Questo funziona per tipi differenti di redirezione.

Esercizio: Si analizzi lo script seguente.

```
#! /usr/bin/env bash
mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 &
exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)
exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr \
                    | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 &
   exec 3> /tmp/fifo2

   echo 1st, allo stdout
   sleep 1
   echo 2nd, allo stderr >&2
   sleep 1
   echo 3rd, a fd 3 >&3
   sleep 1
   echo 4th, a fd 4 >&4
   sleep 1
   echo 5th, to fd 5 >&5
   sleep 1
   echo 6th, tramite una pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
   sleep 1
   echo 7th, a fd 6 >&6
   sleep 1
   echo 8th, a fd 7 >&7
   sleep 1
   echo 9th, a fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-


rm -f /tmp/fifo1 /tmp/fifo2


# Per ogni comando e subshell, indicate il fd in uso e a cosa punta.
```

```
exit 0
```

Appendix F. Localizzazione

La localizzazione è una funzionalità di Bash non documentata.

Uno script di shell localizzato visualizza il testo dell'output nella lingua che è stata definita, nel sistema, come locale. Un utente Linux di Berlino, Germania, preferirebbe gli output degli script in tedesco, mentre suo cugino di Berlin, Maryland, li vorrebbe in inglese.

Per creare uno script localizzato, che visualizzi nella lingua dell'utente tutti i messaggi (messaggi d'errore, prompt, ecc,), si usi lo schema seguente.

```
#!/bin/bash
# localized.sh
# Script di Stephane Chazelas, modificato da Bruno Haible

.gettext.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
}

cd $var || error "‘eval_gettext \"Can’t cd to \$var.\"’"
read -p "‘gettext \"Enter the value: \"’" var
# ...
```

```
bash$ bash -D localized.sh
"Can’t cd to %s."
 "Enter the value: "
```

Così viene elencato tutto il testo localizzato. (L'opzione `-D` elenca le stringhe tra doppi apici, precedute da $, senza eseguire lo script.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
msgid "Can’t cd to %s."
msgstr ""
#: a:7
msgid "Enter the value: "
msgstr ""
```

L'opzione di Bash `--dump-po-strings` assomiglia all'opzione `-D`, ma usa il formato gettext "po".

> **Note:** Bruno Haible precisa:
>
> A partire da gettext-0.12.2, si raccomanda l'uso di **xgettext -o - localized.sh** invece di **bash --dump-po-strings localized.sh**, perchè **xgettext** . . .

1. interpreta i comandi gettext e eval_gettext (mentre bash --dump-po-strings interpreta solamente la sua deprecata sintassi $"...")

2. può togliere i commenti che il programmatore ha inserito per dare informazioni al traduttore.

Questo codice di shell, quindi, non è più specifico di Bash: funziona allo stesso modo anche con Bash 1.x e altre implementazioni /bin/sh.

Ora è necessario creare un file `linguaggio.po`, per ogni lingua in cui si vuole vengano tradotti i messaggi degli script, specificando *`msgstr`*. Ad esempio:

fr.po:

```
#: a:6
msgid "Can’t cd to %s."
msgstr "Impossible de se positionner dans le répertoire %s."
#: a:7
msgid "Enter the value: "
msgstr "Entrez la valeur : "
```

Quindi si esegue msgfmt.

`msgfmt -o localized.sh.mo fr.po`

Il risultante file `localized.sh.mo` va collocato nella directory `/usr/local/share/locale/fr/LC_MESSAGES` e, all’inizio dello script, si inseriscono le righe:

```
TEXTDOMAINDIR=/usr/local/share/locale
TEXTDOMAIN=localized.sh
```

Se un utente di un sistema francese dovesse eseguire lo script, otterrebbe i messaggi nella sua lingua.

> **Note:** Nelle versioni più vecchie di Bash, o in altre shell, la localizzazione richiede l’uso di gettext con l’opzione `-s`. In questo caso, lo script viene così modificato:
>
> ```
> #!/bin/bash
> # localized.sh
>
> E_CDERROR=65
>
> error() {
>   local format=$1
>   shift
>   printf "$(gettext -s "$format")" "$@" >&2
>   exit $E_CDERROR
> }
> cd $var || error "Can’t cd to %s." "$var"
> read -p "$(gettext -s "Enter the value: ")" var
> # ...
> ```

Le variabili `TEXTDOMAIN` e `TEXTDOMAINDIR` devono essere esportate.

Appendice scritta da Stephane Chazelas, con modifiche suggerite da Bruno Haible, manutentore di gettext GNU.

Appendix G. Cronologia dei comandi

La shell Bash dispone di strumenti da riga di comando per gestire e manipolare la *cronologia dei comandi* dell'utente. Si tratta, innanzi tutto, di una comodità, un mezzo per evitare la continua ridigitazione di comandi.

Comandi di cronologia di Bash:

1. **history**
2. **fc**

```
bash$ history
   1  mount /mnt/cdrom
   2  cd /mnt/cdrom
   3  ls
    ...
```

Le variabili interne associate a questi comandi sono:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. !!
8. !$
9. !#
10. !N
11. !-N
12. !STRING
13. !?STRING?
14. ^STRING^string^

Purtroppo, questi strumenti non possono essere usati negli script di Bash.

```
#!/bin/bash
# history.sh
# Tentativo di usare il comando 'history' in uno script.
```

```
history

# Lo script non produce alcun output.
# Inseriti negli script, i comandi di cronologia non funzionano.
```

```
bash$ ./history.sh
(nessun output)
```

Appendix H. Un esempio di file `.bashrc`

Il file `~/.bashrc` determina il comportamento delle shell interattive. Un attento esame di questo file porta ad una migliore comprensione di Bash.

Emmanuel Rouat (mailto:emmanuel.rouat@wanadoo.fr) ha fornito il seguente, molto elaborato, file `.bashrc`, scritto per un sistema Linux. Egli gradirebbe, anche, commenti ed opinioni da parte dei lettori.

Lo si studi attentamente, sapendo che si è liberi di riutilizzarne frammenti di codice e funzioni nei propri file `.bashrc` o anche negli script.

Example H-1. Un semplice file `.bashrc`

```
#=============================================================
#
# PERSONAL $HOME/.bashrc FILE for bash-2.05 (or later)
#
# This file is read (normally) by interactive shells only.
# Here is the place to define your aliases, functions and
# other interactive features like your prompt.
#
# This file was designed (originally) for Solaris.
# --> Modified for Linux.
# This bashrc file is a bit overcrowded - remember it is just
# just an example. Tailor it to your needs
#
#=============================================================

# --> Comments added by HOWTO author.

#-----------------------------------
# Source global definitions (if any)
#-----------------------------------

if [ -f /etc/bashrc ]; then
        . /etc/bashrc   # --> Read /etc/bashrc, if present.
fi

#--------------------------------------------------------------
# Automatic setting of $DISPLAY (if not set already)
# This works for linux and solaris - your mileage may vary....
#--------------------------------------------------------------

if [ -z ${DISPLAY:=""} ]; then
    DISPLAY=$(who am i)
    DISPLAY=${DISPLAY%%\!*}
    if [ -n "$DISPLAY" ]; then
        export DISPLAY=$DISPLAY:0.0
    else
        export DISPLAY=":0.0"  # fallback
    fi
fi
```

```
#---------------
# Some settings
#---------------

set -o notify
set -o noclobber
set -o ignoreeof
set -o nounset
#set -o xtrace          # useful for debuging

shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s histappend histreedit
shopt -s extglob        # useful for programmable completion

#-----------------------
# Greeting, motd etc...
#-----------------------

# Define some colors first:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'              # No Color
# --> Nice. Has the same effect as using "ansi.sys" in DOS.

# Looks best on a black background.....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} - DISPLAY on ${RED}$DISPLAY${NC}\n"
date
if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s     # makes our day a bit more fun.... :-)
fi

function _exit()        # function to run upon exit of shell
{
    echo -e "${RED}Hasta la vista, baby${NC}"
}
trap _exit 0

#---------------
# Shell prompt
#---------------

function fastprompt()
```

```
{
    unset PROMPT_COMMAND
    case $TERM in
        *term | rxvt )
            PS1="[\h] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        *)
            PS1="[\h] \W > " ;;
    esac
}

function powerprompt()
{
    _powerprompt()
    {
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
        TIME=$(date +%H:%M)
    }

    PROMPT_COMMAND=_powerprompt
    case $TERM in
        *term | rxvt  )
            PS1="${cyan}[\$TIME \$LOAD]$NC\n[\h \#] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        linux )
            PS1="${cyan}[\$TIME - \$LOAD]$NC\n[\h \#] \w > " ;;
        * )
            PS1="[\$TIME - \$LOAD]\n[\h \#] \w > " ;;
    esac
}

powerprompt     # this is the default prompt - might be slow
                # If too slow, use fastprompt instead....

#============================================================
#
# ALIASES AND FUNCTIONS
#
# Arguably, some functions defined here are quite big
# (ie 'lowercase') but my workstation has 512Meg of RAM, so .....
# If you want to make this file smaller, these functions can
# be converted into scripts.
#
# Many functions were taken (almost) straight from the bash-2.04
# examples.
#
#============================================================

#-------------------
# Personnal Aliases
#-------------------

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

```
# -> Prevents accidentally clobbering files.

alias h='history'
alias j='jobs -l'
alias r='rlogin'
alias which='type -all'
alias ..='cd ..'
alias path='echo -e ${PATH//:/\\n}'
alias print='/usr/bin/lp -o nobanner -d $LPDEST'   # Assumes LPDEST is defined
alias pjet='enscript -h -G -fCourier9 -d $LPDEST'  # Pretty-print using enscript
alias background='xv -root -quit -max -rmode 5' # put a picture in the background
alias vi='vim'
alias du='du -h'
alias df='df -kh'

# The 'ls' family (this assumes you use the GNU ls)
alias ls='ls -hF --color' # add colors for filetype recognition
alias lx='ls -lXB' # sort by extension
alias lk='ls -lSr' # sort by size
alias la='ls -Al' # show hidden files
alias lr='ls -lR' # recursice ls
alias lt='ls -ltr' # sort by date
alias lm='ls -al |more' # pipe through 'more'
alias tree='tree -Cs' # nice alternative to 'ls'


# tailoring 'less'
alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-' # Use this if lesspipe.sh exists
export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# spelling typos - highly personnal :-)
alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'

#----------------
# a few fun ones
#----------------

function xtitle ()
{
    case $TERM in
        *term | rxvt)
            echo -n -e "\033]0;$*\007" ;;
        *)  ;;
    esac
}
```

```
# aliases...
alias top=’xtitle Processes on $HOST && top’
alias make=’xtitle Making $(basename $PWD) ; make’
alias ncftp="xtitle ncFTP ; ncftp"

# .. and functions
function man ()
{
    xtitle The $(basename $1|tr -d .[:digit:]) manual
    man -a "$*"
}

function ll(){ ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -v "^d|total "; }
function xemacs() { { command xemacs -private $* 2>&- & } && disown ;}
function te()  # wrapper around xemacs/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( xemacs "$@" & );
    fi
}

#----------------------------------
# File & strings related functions:
#----------------------------------

function ff() { find . -name ’*’$1’*’ ; }                 # find a file
function fe() { find . -name ’*’$1’*’ -exec $2 {} \; ; }  # find a file and run $2 on it
function fstr() # find a string in a set of files
{
    if [ "$#" -gt 2 ]; then
        echo "Usage: fstr \"pattern\" [files] "
        return;
    fi
    SMSO=$(tput smso)
    RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print | xargs grep -sin "$1" | \
sed "s/$1/$SMSO$1$RMSO/gI"
}

function cuttail() # cut last n lines in file, 10 by default
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase()  # move filenames to lowercase
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
```

```
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # swap 2 filenames around
{
    local TMPFILE=tmp.$$
    mv $1 $TMPFILE
    mv $2 $1
    mv $TMPFILE $2
}

#-------------------------------------
# Process/system related functions:
#-------------------------------------

function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# This function is roughly the same as 'killall' on linux
# but has no equivalent (that I know of) on Solaris
function killps()   # kill by process name
{
    local pid pname sig="-TERM"   # default signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
echo "Usage: killps [-SIGNAL] pattern"
return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
if ask "Kill process $pid <$pname> with signal $sig?"
    then kill $sig $pid
fi
    done
}

function my_ip() # get IP adresses
{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -e s/P-t-P://)
}
```

```
function ii()   # get current host related info
{
    echo -e "\nYou are logged on ${RED}$HOST"
    echo -e "\nAdditionnal information:$NC " ; uname -a
    echo -e "\n${RED}Users logged on:$NC " ; w -h
    echo -e "\n${RED}Current date :$NC " ; date
    echo -e "\n${RED}Machine stats :$NC " ; uptime
    echo -e "\n${RED}Memory stats :$NC " ; free
    my_ip 2>&- ;
    echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-"Not connected"}
    echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
    echo
}


# Misc utilities:

function repeat()       # repeat n times command
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> C-like syntax
        eval "$@";
    done
}


function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#============================================================================
#
# PROGRAMMABLE COMPLETION - ONLY SINCE BASH-2.04
# (Most are taken from the bash 2.05 documentation)
# You will in fact need bash-2.05 for some features
#
#============================================================================

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
    echo "You will need to upgrade to version 2.05 for programmable completion"
    return
fi

shopt -s extglob        # necessary
set +o nounset # otherwise some completions will fail

complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
```

```
complete -A command    nohup exec eval trace gdb
complete -A command    command type which
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # currently same as builtins
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

complete -f -d -X '*.gz'  gzip
complete -f -d -X '*.bz2' bzip2
complete -f -o default -X '!*.gz'  gunzip
complete -f -o default -X '!*.bz2' bunzip2
complete -f -o default -X '!*.pl'  perl perl5
complete -f -o default -X '!*.ps'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.+(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(jpg|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.mp3' mpg123
complete -f -o default -X '!*.ogg' ogg123


# This is a 'universal' completion function - it works when commands have
# a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'
_universal_func ()
{
    case "$2" in
-*) ;;
*) return ;;
    esac

    case "$1" in
\~*) eval cmd=$1 ;;
*) cmd="$1" ;;
    esac
    COMPREPLY=( $("$cmd" --help | sed  -e '/--/!d' -e 's/.*--\([^ ]*\).*/--\1/'| \
grep ^"$2" |sort -u) )
}
complete  -o default -F _universal_func ldd wget bash id info
```

```
_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # if prev argument is -f, return possible filename completions.
    # we could be a little smarter here and return matches against
    # 'makefile Makefile *.mk', whatever exists
    case "$prev" in
        -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # if we want an option, return the possible posix options
    case "$cur" in
        -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac

    # make reads 'makefile' before 'Makefile'
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk               # local convention
    fi

    # before we scan for targets, see if a makefile name was specified
    # with -f
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -*f ]]; then
            eval makef=${COMP_WORDS[i+1]}       # eval for tilde expansion
            break
        fi
    done

        [ -z "$makef" ] && makef=$mdef

    # if we have a partial word to complete, restrict completions to
    # matches of that word
    if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi

    # if we don't want to use *.mk, we can take out the cat and use
    # test -f $makef and input redirection
    COMPREPLY=( $(cat $makef 2>/dev/null | awk 'BEGIN {FS=":"} /^[^.#   ][^=]*:/ {print $1}' | tr -s
}

complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake

_configure_func ()
{
```

```
    case "$2" in
        -*)     ;;
        *)      return ;;
    esac

    case "$1" in
        \~*)    eval cmd=$1 ;;
        *)      cmd="$1" ;;
    esac

    COMPREPLY=( $("$cmd" --help | awk '{if ($1 ~ /--.*/) print $1}' | grep ^"$2" | sort -u) )
}

complete -F _configure_func configure

# cvs(1) completion
_cvs ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
export history import log rdiff release remove rtag status \
tag update' $cur ))
    else
COMPREPLY=( $( compgen -f $cur ))
    fi
    return 0
}
complete -F _cvs cvs


_killall ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # get a list of processes (the first sed evaluation
    # takes care of swapped out processes, the second
    # takes care of getting the basename of the process)
    COMPREPLY=( $( /usr/bin/ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}

complete -F _killall killall killps
```

```
# Local Variables:
# mode:shell-script
# sh-shell:bash
# End:
```

Appendix I. Conversione dei file batch di DOS in script di shell

Un certo numero di programmatori ha imparato lo scripting su PC con sistema operativo DOS. Anche il frammentario linguaggio dei file batch di DOS consente di scrivere delle applicazioni e degli script piuttosto potenti, anche se questo richiede un ampio impiego di espedienti e stratagemmi. Talvolta, la necessità spinge a convertire vecchi file batch di DOS in script di shell UNIX. Questa operazione, generalmente, non è difficile, dal momento che gli operatori dei file batch DOS sono in numero inferiore rispetto agli analoghi operatori dello scripting di shell.

Table I-1. Parole chiave / variabili / operatori dei file batch e loro equivalenti di shell

Operatore di File Batch	Corrispondente di scripting di shell	Significato
`%`	$	prefisso dei parametri da riga di comando
`/`	-	prefisso per le opzione di un comando
`\`	/	separatore di percorso
`==`	=	(uguale a) verifica di confronto di stringhe
`!==!`	!=	(non uguale a) verifica di confronto di stringhe
`|`	|	pipe
`@`	set +v	non visualizza il comando corrente
`*`	*	“carattere jolly” per nomi di file
`>`	>	redirezione di file (sovrascrittura)
`>>`	>>	redirezione di file (accodamento)
`<`	<	redirezione dello `stdin`
`%VAR%`	$VAR	variabile d’ambiente
`REM`	#	commento
`NOT`	!	nega la verifica successiva
`NUL`	`/dev/null`	“buco nero” dove seppellire l’output dei comandi
`ECHO`	echo	visualizzazione (molte più opzioni in Bash)
`ECHO.`	echo	visualizza una riga vuota
`ECHO OFF`	set +v	non visualizza il/i comando/i successivo/i
`FOR %%VAR IN (LISTA) DO`	for var in [lista]; do	ciclo “for”
`:ETICHETTA`	nessuno (non necessario)	etichetta
`GOTO`	nessuno (usa una funzione)	salta ad un altra parte dello script
`PAUSE`	sleep	pausa o intervallo di attesa

Operatore di File Batch	Corrispondente di scripting di shell	Significato
`CHOICE`	case o select	menu di scelta
`IF`	if	condizione if
`IF EXIST` *`NOMEFILE`*	if [-e nomefile]	verifica l'esistenza del file
`IF !%N==!`	if [-z "$N"]	verifica se il parametro "N" non è presente
`CALL`	source o . (operatore punto)	"include" un altro script
`COMMAND /C`	source o . (operatore punto)	"include" un altro script (uguale a CALL)
`SET`	export	imposta una variabile d'ambiente
`SHIFT`	shift	scorrimento a sinistra dell'elenco degli argomenti da riga di comando
`SGN`	-lt o -gt	segno (di intero)
`ERRORLEVEL`	$?	exit status
`CON`	`stdin`	"console" (`stdin`)
`PRN`	`/dev/lp0`	dispositivo di stampa (generico)
`LPT1`	`/dev/lp0`	primo dispositivo di stampa
`COM1`	`/dev/ttyS0`	prima porta seriale

Ovviamente, i file batch contengono, di solito, comandi DOS. Per una corretta conversione anche questi devono essere sostituiti con i loro equivalenti UNIX.

Table I-2. Comandi DOS e loro equivalenti UNIX

Comando DOS	Corrispettivo UNIX	Effetto
`ASSIGN`	ln	collega file o directory
`ATTRIB`	chmod	cambia i permessi del file
`CD`	cd	cambia directory
`CHDIR`	cd	cambia directory
`CLS`	clear	pulisce lo schermo
`COMP`	diff, comm, cmp	confronta i file
`COPY`	cp	copia i file
`Ctl-C`	Ctl-C	interruzione (segnale)
`Ctl-Z`	Ctl-D	EOF (end-of-file)
`DEL`	rm	cancella il/i file
`DELTREE`	rm -rf	cancella ricorsivamente una directory
`DIR`	ls -l	elenca una directory
`ERASE`	rm	cancella il/i file
`EXIT`	exit	esce dal processo corrente
`FC`	comm, cmp	confronta i file
`FIND`	grep	ricerca le stringhe nei file

Comando DOS	Corrispettivo UNIX	Effetto
MD	mkdir	crea una directory
MKDIR	mkdir	crea una directory
MORE	more	filtro per l'impaginazione del testo del file
MOVE	mv	spostamento
PATH	$PATH	percorso degli eseguibili
REN	mv	rinomina (sposta)
RENAME	mv	rinomina (sposta)
RD	rmdir	cancella una directory
RMDIR	rmdir	cancella una directory
SORT	sort	ordina il file
TIME	date	visualizza l'ora di sistema
TYPE	cat	visualizza il file allo `stdout`
XCOPY	cp	copia (estesa) di file

> **Note:** In pratica, tutti gli operatori e i comandi di shell, e UNIX, possiedono molte più opzioni e funzionalità rispetto ai loro equivalenti DOS e dei file batch. Inoltre, molti file batch di DOS si basano su utility ausiliarie, come **ask.com**, una farraginosa controparte di read.
>
> DOS supporta una serie molto limitata e incompatibile di caratteri jolly per l'espansione dei nomi dei file, riconoscendo solo i caratteri * e ?.

Convertire un file batch di DOS in uno script di shell è, solitamente, semplice ed il risultato, molte volte, è più leggibile dell'originale.

Example I-1. VIEWDATA.BAT: file batch DOS

```
REM VIEWDATA

REM ISPIRATO DA UN ESEMPIO PRESENTE IN "DOS POWERTOOLS"
REM                                    DI PAUL SOMERSON


@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  SE NON CI SONO ARGOMENTI DA RIGA DI COMANDO...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  VISUALIZZA LA RIGA DELLA STRINGA VERIFICATA, QUINDI ESCE.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  VISUALIZZA L'INTERO FILE, 1 PAGINA ALLA VOLTA.
```

```
:EXIT0
```

La conversione dello script rappresenta un miglioramento.

Example I-2. viewdata.sh: Script di shell risultante dalla conversione di VIEWDATA.BAT

```
#!/bin/bash
# Conversione di VIEWDATA.BAT in script di shell.

FILEDATI=/home/bozo/datafiles/book-collection.data
NRARG=1

# @ECHO OFF       In questo caso il comando è inutile.

if [ $# -lt "$NRARG" ]    # IF !%1==! GOTO VIEWDATA
then
  less $FILEDATI          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $FILEDATI     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                    # :EXIT0

# Non sono necessari GOTO, etichette, giochi di specchi e imbrogli.
# Il risultato della conversione è uno script breve, dolce e pulito,
# il che non può dirsi dell'originale.
```

In Shell Scripts on the PC (http://www.maem.umr.edu/~batch/), sul sito di Ted Davis, è presente un'ampia serie di manuali sull'arte, ormai fuori moda, della programmazione di file batch. È immaginabile che alcune delle sue ingegnose tecniche possano aver rilevanza anche per gli script di shell.

Appendix J. Esercizi

J.1. Analisi di script

Si esamini lo script seguente. Lo si esegua e, quindi, si spieghi quello che fa. Si commenti lo script e lo si riscriva in modo che risulti più compatto ed elegante.

```
#!/bin/bash

MAX=10000


  for((nr=1; nr<$MAX; nr++))
  do

    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
      continue
    fi

  break   # Cosa succede se si commenta questa riga? E perché?

  done

  echo "Numero = $nr"


exit 0
```

Un lettore ha inviato il seguente frammento di codice.

```
while read RIGA
do
  echo $RIGA
done < ‘tail -f /var/log/messages‘
```

Il suo desiderio era quello di scrivere uno script che memorizzasse le modifiche del file di log di sistema `/var/log/messages`. Sfortunatamente, il precedente codice si blocca e non fa niente di utile. Perché? Si risolva il problema in modo che funzioni correttamente (Suggerimento: invece di redirigere lo `stdin` del ciclo, si provi con una pipe.)

Si analizzi l'Example A-11 e lo si riorganizzi in uno stile più semplice e logico. Si veda quante delle variabili in esso presenti possono essere eliminate e lo si ottimizzi per aumentarne la velocità d'esecuzione.

Si modifichi lo script in modo che accetti, come input, un qualsiasi file di testo ASCII per la sua "generazione" iniziale. Lo script dovrà leggere i primi caratteri `$ROW*$COL` ed impostare le occorrenze delle vocali come celle "vive". Suggerimento: ci si accerti di aver trasformato tutti gli spazi presenti nel file di input in caratteri di sottolineatura.

J.2. Scrivere script

Per ciascuno dei compiti sotto elencati, si scriva uno script che svolga correttamente quanto richiesto.

Facili

Elenco della directory home

Si esegua un elenco ricorsivo della home directory dell'utente e si salvino le informazioni in un file. Questo file va compresso e lo script deve visualizzare un messaggio che invita l'utente ad inserire un dischetto e a premere, successivamente, il tasto **INVIO**. Alla fine il file risulterà essere registrato sul floppy.

Modifica dei cicli for in cicli while e until

Si sostituiscano i *cicli for* presenti in Example 10-1 con *cicli while*. Suggerimento: si registrino i dati in un array, quindi si passino in rassegna gli elementi dell'array stesso.

Essendo "il più" già fatto, ora si convertano i cicli dell'esempio in *cicli until*.

Modifica dell'interlinea di un file di testo

Si scriva uno script che legga ciascuna riga del file indicato e la visualizzi allo `stdout`, ma seguita da una riga bianca aggiuntiva. Questo produrrà, come risultato, un file con *interlinea doppia*.

Si aggiunga il codice necessario affinché venga effettuato un controllo sui necessari argomenti che devono essere passati allo script da riga di comando (il nome di un file) e per verificare che il file esista.

Una volta certi che lo script funzioni correttamente, lo si modifichi in modo da ottenere una *interlinea tripla* del file indicato.

Infine, si scriva uno script che rimuova tutte le righe vuote dal file indicato in modo che il testo risulti composto con *interlinea singola*.

Elenco rovesciato

Si scriva uno script che si autovisualizzi allo `stdout`, ma in *senso inverso* (prima l'ultima riga, poi la penultima, ecc.).

Decompressione automatica di file

Dato come input un elenco di file, questo script interrogherà ciascun file (verificando l'output del comando file) per controllare quale tipo di compressione è stata ad esso applicata. Lo script, quindi, dovrà invocare automaticamente l'appropriato comando di decompressione (**gunzip**, **bunzip2**, **unzip**, **uncompress** o altro). Se, tra i file indicati, ve ne dovessero essere di non compressi, lo script dovrà visualizzare un messaggio d'avvertimento e non effettuare, su tali file, nessun'altra azione.

ID unico di sistema

Si generi un numero identificativo "unico", di sei cifre esadecimali, per il vostro computer. *Non* si usi l'inadeguato comando hostid. Suggerimento: **md5sum `/etc/passwd`** e quindi si scelgano le prime sei cifre dell'output.

Backup

Si archivino come "tarball" (file `*.tar.gz`) tutti i file presenti nella vostra directory home (`/home/vostro-nome`) che sono stati modificati nelle ultime 24 ore. Suggerimento: si usi find.

Primi

Si visualizzino (allo stdout) tutti i numeri primi compresi tra 60000 e 63000. L'output dovrebbe essere elegantemente ordinato in colonne (suggerimento: si usi printf).

Numeri della lotteria

Un tipo di lotteria prevede l'estrazione di cinque numeri diversi nell'intervallo 1 - 50. Si scriva uno script che generi cinque numeri pseudocasuali compresi in quell'intervallo, *senza duplicazioni*. Lo script dovrà dare la possibilità di scelta tra la visualizzazione dei numeri allo `stdout` o il loro salvataggio in un file, con data e ora in cui quella particolare serie numerica è stata generata.

Intermedi

Gestione dello spazio su disco

Si elenchino, uno alla volta, tutti i file della directory `/home/nomeutente` di dimensioni maggiori di 100K. Ad ogni file elencato si dovrà dare all'utente la possibilità di scelta tra la sua cancellazione o la sua compressione, dopo di che verrà visualizzato il file successivo. Si registrino in un file di log i nomi di tutti i file cancellati nonché la data e l'ora della cancellazione.

Cancellazione sicura

Si scriva, in forma di script, il comando per la cancellazione di "sicurezza" `srm.sh`. I file, passati allo script come argomenti da riga di comando, non verranno cancellati, ma, se non già compressi, dovranno esserlo tramite gzip (per la verifica si usi file), e quindi spostati nella directory `/home/nomeutente/trash`. Al momento dell'invocazione, lo script controllerà nella directory "trash" i file in essa presenti da più di 48 ore e li cancellerà.

Scambiare soldi

Qual'è la via più efficiente per scambiare $ 1.68, usando il minor numero di monete correntemente in circolazione (fino a 25 cents)? 6 quarti di dollaro, 1 dime (moneta da dieci centesimi di dollaro), un nickel (moneta da 5 centesimi) e tre monete da un centesimo.

Dato come input, da riga di comando, un importo arbitrario espresso in dollari e centesimi ($*.??), si calcoli come scambiarlo utilizzando il minor numero di monete. Se il vostro paese non sono gli Stati Uniti, si può utilizzare la valuta locale. Lo script dovrà verificare l'input e, quindi, trasformarlo in multipli dell'unità monetaria più piccola (centesimi o altro). Suggerimento: si dia un'occhiata a Example 23-4.

Equazioni quadratiche

Si risolva un'equazione "quadratica" nella forma *Ax^2 + Bx + C = 0*. Lo script dovrà avere come argomenti i coefficienti **A**, **B** e **C**, e restituire il risultato con quattro cifre decimali.

Suggerimento: si colleghino i coefficienti, con una pipe, a bc, utilizzando la ben nota formula *x = (-B +/- sqrt(B^2 - 4AC)) / 2A*.

Somma di numeri di corrispondenza

Si calcoli la somma di tutti i numeri di cinque cifre (compresi nell'intervallo 10000 - 99999) che contengono *esattamente due*, e solo due, cifre della serie seguente: { 4, 5, 6 }. All'interno dello stesso numero, queste possono ripetersi, nel qual caso la stessa cifra non può apparire più di due volte.

Alcuni esempi di numeri di corrispondenza che soddisfano il criterio più sopra enunciato sono 42057, 74638 e 89515.

Numeri fortunati

Un "numero fortunato" è quello in cui la somma, per addizioni successive, delle cifre che lo compongono dà come risultato 7. Per esempio, 62431 è un "numero fortunato" (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Si ricerchino tutti i "numeri fortunati" compresi tra 1000 e 10000.

Ordinare alfabeticamente una stringa

Si pongano in ordine alfabetico (in ordine ASCII) le lettere di una stringa arbitraria passata da riga di comando.

Verifica

Si verifichi il file `/etc/passwd` e se ne visualizzi il contenuto in un preciso, e facilmente comprensibile, formato tabellare.

Visualizzazione intelligente di un file dati

Alcuni programmi per database e fogli di calcolo sono soliti salvare i propri file usando, *come separatore di campo*, la virgola (CSV - comma-separated values)). Spesso, altre applicazioni hanno la necessità di accedere a questi file.

Dato un file con queste caratteristiche, nella forma:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
```

. . .

si riorganizzino i dati e li si visualizzi allo `stdout` in colonne intestate e correttamente distanziate.

Giustificazione

Dato come input un testo ASCII, fornito o dallo `stdin` o da un file, si agisca sulla spaziatura delle parole di ogni riga, con giustificazione a destra, in modo che la stessa corrisponda alle dimensioni specificate dall'utente, inviando, successivamente, il risultato allo `stdout`.

Mailing List

Usando il comando mail, si scriva uno script che gestisca una semplice mailing list. Questo script dovrà spedire automaticamente, per e-mail, un'informativa mensile della società, il cui testo viene preso dal file indicato, a tutti gli indirizzi presenti nella mailing list, che lo script ricaverà da un altro file specificato.

Creare password

Si generino password di 8 caratteri (compresi negli intervalli [0-9], [A-Z], [a-z]) pseudocasuali. Ogni password dovrà contenere almeno due cifre.

Difficili

Verifica delle password

Si scriva uno script che controlli e convalidi le password. Lo scopo è quello di segnalare le password "deboli" o che possono essere facilmente indovinate.

Allo script deve essere passata una password di prova, come parametro da riga di comando. Per essere considerata valida, una password deve avere i seguenti requisiti minimi:

- Lunghezza minima di 8 caratteri
- Deve contenere almeno un carattere numerico
- Deve contenere almeno uno dei seguenti caratteri non alfabetici: @, #, $, %, &, *, +, -, =

Facoltativi:

- Eseguire un controllo di dizionario su tutte le sequenze di almeno quattro caratteri alfabetici consecutivi presenti nella password in verifica. Questo per eliminare quelle password contenenti "parole" che si possono trovare in un normale dizionario.
- Permettere allo script di controllare tutte le password presenti sul sistema. Possano risiedere, o meno, nel file `/etc/passwd`.

Questo esercizio richiede la perfetta padronanza delle Espressioni Regolari.

Log degli accessi ai file

Si crei un file di log degli accessi ai file presenti in `/etc` avvenuti nel corso della giornata. Le informazioni devono comprendere: il nome del file, il nome dell'utente, l'ora di accesso. Si dovrà anche contrassegnare quel/quei file che hanno subito delle modifiche. Questi dati dovranno essere registrati nel file di log in record ben ordinati.

Controllo dei processi

Lo script deve controllare in continuazione tutti i processi in esecuzione e annotare quanti processi figli sono stati generati da ciascun processo genitore. Se un processo genera più di cinque processi figli, allora lo script deve spedire una e-mail all'amministratore di sistema (o a root) con tutte le informazioni di maggior importanza, tra cui l'ora, il PID del processo genitore, i PID dei processi figli, etc. Lo script deve anche scrivere un rapporto in un file di log ogni dieci minuti.

Togliere i commenti

Si tolgano tutti i commenti da uno script di shell il cui nome andrà specificato da riga di comando. Si faccia attenzione a non cancellare la "riga #!".

Conversione HTML

Si converta un dato file di testo nel formato HTML. Questo script non interattivo dovrà inserire automaticamente tutti gli appropriati tag HTML nel file specificato come argomento.

Togliere i tag HTML

Si tolgano tutti i tag da un file HTML specificato, quindi lo si ricomponga in righe di dimensione compresa tra i 60 e i 75 caratteri. Si reimpostino appropriatamente i paragrafi e le spaziature dei blocchi di testo, e si convertano le tabelle HTML in quelle approssimativamente corrispondenti del formato testo.

Conversione XML

Si converta un file XML sia nel formato HTML che in formato testo.

Caccia agli spammer

Si scriva uno script che analizzi una e-mail di spam eseguendo una ricerca DNS sugli indirizzi IP presenti nell'intestazione del messaggio, per identificare i veri host così come l'ISP d'origine. Lo script dovrà reindirizzare il messaggio di spam inalterato agli ISP responsabili. Naturalmente, sarà necessario togliere *l'indirizzo IP del proprio provider* per non finire col lamentarsi con se stessi.

Se necessario, si usino gli appropriati comandi per l'analisi di rete.

Codice Morse

Si converta un file di testo in codice Morse. Ogni carattere del testo verrà rappresentato dal corrispondente carattere dell'alfabeto Morse, formato da punti e linee (si usi il trattino di sottolineatura), separato l'uno dall'altro da spazi. Per esempio, "script" ===> "... _._. ._. .. .__. _".

Editor esadecimale

Si esegua una visualizzazione in esadecimale di un file binario specificato come argomento. L'output dovrà avere i campi ben ordinati in forma tabellare, con il primo campo che indica l'indirizzo di memoria, ciascuno

dei successivi otto campi un numero esadecimale di 4 byte e l'ultimo campo l'equivalente ASCII dei precedenti otto campi.

Simulare uno scorrimento del registro

Ispirandosi all'Example 26-14, si scriva uno script che simuli uno shift di registro a 64 bit, in forma di array. Si implementino le funzioni per il *caricamento* del registro, per lo *scorrimento a sinistra* e per lo *scorrimento a destra*. Infine, si scriva una funzione che interpreti il contenuto del registro come caratteri ASCII di otto per otto bit.

Determinante

Si risolva una determinante 4 x 4.

Parole nascoste

Si scriva un generatore di puzzle di "parole", vale a dire uno script che celi 10 parole fornite come input in una matrice 10 x 10 di lettera casuali. Le parole possono essere inserite orizzontalmente, verticalmente o diagonalmente.

Anagrammare

Si trovino gli anagrammi di un input di quattro lettere. Ad esempio, gli anagrammi di *word* sono: *do or rod row word*. Come elenco di riferimento si può utilizzare `/usr/share/dict/linux.words`.

Indice di comprensione

L'"indice di comprensione" di un brano di un testo, indica la difficoltà di lettura dello stesso per mezzo di un numero che corrisponde, approssimativamente, al livello di scolarizzazione. Ad esempio, un brano con indice 12 dovrebbe essere capito da tutti coloro che hanno avuto dodici anni di scolarizzazione.

La versione Gunning dell'indice di comprensione usa il seguente algoritmo.

1. Si sceglie un brano, di almeno cento parole, da un testo.
2. Si conta il numero delle frasi (la parte di frase che è stata troncata perché alla fine del brano, si calcola come se fosse intera).
3. Si calcola il numero medio di parole per frase.

   MEDIA_PAROLE = PAROLE_TOTALI / NUMERO_FRASI

4. Si conta il numero delle parole "difficili" presenti nel brano -- quelle formate da almeno tre sillabe. Questa quantità viene divisa per il totale delle parole, per ottenere la proporzione di parole difficili.

   PRO_PAROLE_DIFFICILI = PAROLE_LUNGHE / TOTALE_PAROLE

5. L'indice di comprensione Gunning risulta dalla somma delle due precedenti grandezze moltiplicata per 0.4, con arrotondamento all'intero più prossimo.

   INDICE_GUNNING = int (0.4 * (MEDIA_PAROLE + PRO_PAROLE_DIFFICILI))

Il passaggio nr. 4 è la parte di gran lunga più difficile dell'esercizio. Esistono diversi algoritmi per il conteggio delle sillabe di una parola. Una formula empirica approssimativa potrebbe prendere in considerazione il numero di lettere che compongono la parola e il modo in cui le consonanti e le vocali si alternano.

L'interpretazione restrittiva dell'indice di comprensione Gunning non considera come parole "difficili" le parole composte e i nomi propri, ma questo avrebbe reso lo script enormemente complesso.

Calcolo del PI Greco usando il metodo dell'Ago di Buffon

Il matematico francese del XVIII secolo De Buffon se n'è uscito con un esperimento insolito. Ha fatto cadere ripetutamente un ago di lunghezza "n" su un pavimento di legno, formato da assi lunghe e strette disposte parallelamente. Le linee di giunzione delle assi del pavimento, che sono tutte della stessa larghezza, si trovano, l'una dall'altra, alla distanza fissa "d". Ha annotato il numero totale delle cadute dell'ago nonché il numero di volte in cui lo stesso andava ad intersecare le giunzioni delle assi del pavimento. Il rapporto tra queste due grandezze è risultato essere un multiplo frazionario del PI Greco.

Prendendo come spunto l'Example 12-36, si scriva uno script che esegua una simulazione Monte Carlo dell'Ago di Buffon. Per semplificare le cose, si imposti la lunghezza dell'ago uguale alla distanza tra le giunzioni, $n = d$.

Suggerimento: in verità bisogna tenere in considerazione due variabili critiche: la distanza dal centro dell'ago alla giunzione ad esso più vicina e l'angolo formato dall'ago con quella giunzione. Per l'esecuzione dei calcoli si dovrà usare bc.

Cifrario Playfair

Si implementi in uno script il cifrario Playfair (Wheatstone).

Il cifrario Playfair codifica un testo mediante la sostituzione dei "digrammi" (gruppi di due lettere). Per consuetudine si dovrebbe usare, per la cifratura e la decodifica, una chiave a matrice quadrata di 5 x 5 lettere, poste in un certo ordine.

```
C O D E S
A B F G H
I K L M N
P Q R T U
V W X Y Z
```

```
Ogni lettera alfabetica appare una sola volta, la "I" rappresenta
anche la "J". La parola chiave "CODES", scelta arbitrariamente, viene
per prima e, di seguito, tutte le lettere dell'alfabeto, tralasciando
quelle che formano la parola chiave.

Per la cifratura, si suddivide il messaggio in chiaro in digrammi
(gruppi di 2 lettere). Se un gruppo risulta formato da due lettere
uguali, si cancella la seconda e si forma un nuovo gruppo. Se per
l'ultimo digramma rimane una sola lettera, come seconda si usa un
carattere "nullo", di solito una "X"

QUESTO È UN MESSAGGIO TOP SECRET

QU ES TO EU NM ES SA GI OT OP SE CR ET
```

```
Per ogni digramma vi sono tre possibilità.
---------------------------------------------
1) Entrambe le lettere si trovano su una stessa riga della chiave a
   matrice quadrata. Ciascuna lettera va sostituita con quella che si
   trova immediatamente alla sua destra. Se la lettera da sostituire
   è l'ultima della riga, si userà la prima della stessa riga.

oppure

2) Entrambe le lettere si trovano su una stessa colonna della chiave a
   matrice quadrata. Ciascuna lettera va sostituita con quella che si
   trova immediatamente al di sotto. Se la lettera da sostituire è
   l'ultima della colonna, si userà la prima della stessa colonna.

oppure

3) Entrambe le lettere formano gli angoli di un rettangolo all'interno
   della chiave a matrice quadrata. Ciascuna lettera viene sostituita
   con quella che si trova all'angolo opposto, ma sulla stessa riga.


Il digramma "QU" ricade nel caso nr. 1.
P Q R T U   (Riga che contiene sia "Q" che "U")

Q --> R
U --> P     (si è tornati ad inizio riga)


Il digramma "GI" ricade nel caso nr. 3.
A B F G     (Rettangolo avente "G" e "I" agli angoli)
I K L M

G --> A
I --> M

==========================================================================

Per la decodifica del testo cifrato bisogna invertire, nei casi nr. 1
e nr. 2, la procedura (per la sostituzione ci si sposta nella
direzione opposta). Mentre nulla cambia per quanto riguarda il caso
nr. 3.


Il lavoro, ormai classico, di Helen Fouche Gaines, "Elementary
Cryptoanalysis" (1939), fornisce un resoconto veramente dettagliato
sul Cifrario Playfair e sui relativi metodi di soluzione.
```

Lo script dovrà essere composto da tre sezioni principali

I. Generazione della “chiave a matrice quadrata”, basata su una parola scelta dall’utente.

II. Cifratura del messaggio “in chiaro”.

III. Decodifica del testo cifrato.

Lo script dovrà fare un uso intensivo di array e funzioni.

--

Si è pregati di non inviare all’autore le soluzioni degli esercizi. Vi sono modi migliori per impressionarlo con le proprie abilità, come segnalargli errori e fornirgli suggerimenti per migliorare il libro.

Appendix K. Cronologia delle revisioni

Questo documento è apparso per la prima volta, come HOWTO, nella tarda primavera del 2000. Da allora ha subito numerosi aggiornamenti e revisioni.
Non sarebbe stato possibile realizzare questo libro senza la collaborazione della comunità Linux e, in modo particolare, del Linux Documentation Project (http://www.tldp.org).

Revision History

Revision 0.1 14 giugno 2000 Revised by: mc
Release iniziale.
Revision 0.2 30 ottobre 2000 Revised by: mc
Correzioni, aggiunta di materiale addizionale e script d’esempio.
Revision 0.3 12 febbraio 2001 Revised by: mc
Aggiornamento importante.
Revision 0.4 08 luglio 2001 Revised by: mc
Ulteriori correzioni, molto più materiale e script - una revisione co
Revision 0.5 03 settembre 2001 Revised by: mc
Altro importante aggiornamento. Correzioni, aggiunta di materiale
Revision 1.0 14 ottobre 2001 Revised by: mc
Correzioni, riorganizzazione, aggiunta di materiale. Stable release.
Revision 1.1 06 gennaio 2002 Revised by: mc
Correzioni, aggiunti materiale e script.
Revision 1.2 31 marzo 2002 Revised by: mc
Correzioni, aggiunta di materiale e script.
Revision 1.3 02 giugno 2002 Revised by: mc
’TANGERINE’ release: Piccole correzioni, molto più materiale e ag
Revision 1.4 16 giugno 2002 Revised by: mc
’MANGO’ release: Correzione di diversi errori tipografici, altro mat
Revision 1.5 13 luglio 2002 Revised by: mc

’PAPAYA’ release: Alcune correzioni, molto più materiale e aggiunt
Revision 1.6 29 settembre 2002 Revised by: mc
’POMEGRANATE’ release: qualche correzione, altro materiale, anc
Revision 1.7 05 gennaio 2003 Revised by: mc
’COCONUT’ release: un paio di correzioni, più materiale, ulteriori s
Revision 1.8 10 maggio 2003 Revised by: mc
’BREADFRUIT’ release: diverse correzioni, altro materiale e script.
Revision 1.9 21 giugno 2003 Revised by: mc
’PERSIMMON’ release: correzioni e materiale aggiuntivo.
Revision 2.0 24 agosto 2003 Revised by: mc
’GOOSEBERRY’ release: ampio aggiornamento.
Revision 2.1 14 settembre 2003 Revised by: mc
’HUCKLEBERRY’ release: correzioni ed altro materiale.
Revision 2.2 31 ottobre 2003 Revised by: mc
’CRANBERRY’ release: aggiornamento rilevante.
Revision 2.3 03 gennaio 2004 Revised by: mc
’STRAWBERRY’ release: correzioni e aggiunta di materiale.

Appendix L. Copyright

La “Advanced Bash-Scripting Guide” è sotto copyright © 2000, di Mendel Cooper. L’autore rivendica il copyright anche su tutte le precedenti versioni di questo documento.

Questo esteso copyright riconosce e tutela i diritti di coloro che hanno contribuito alla realizzazione di questo documento.

Questo documento può essere distribuito solo in base ai termini e alle condizioni stabilite dalla Open Publication License (versione 1.0 o successive), http://www.opencontent.org/openpub/. Si applicano, inoltre, i seguenti termini di licenza.

```
A.  È vietata la distribuzione di versioni di questo documento,
    contenenti sostanziali modifiche, senza il consenso esplicito del
    detentore del copyright.

B.  È vietata la distribuzione dell’opera, o derivati
    dell’opera, in forma di libro (cartaceo) di qualsiasi formato,
    senza la preventiva autorizzazione del detentore del copyright.
```

Il precedente *paragrafo A* vieta esplicitamente la *ridefinizione* del documento. Esempio di ridefinizione è l’inserimento del logo di una società o di barre di navigazione in copertina, nella pagina del titolo o nel testo. L’autore acconsente alle seguenti deroghe.

1. Organizzazioni non-profit, come il Linux Documentation Project (http://www.tldp.org) e Sunsite (http://ibiblio.org).
2. Distributori Linux “puri”, come Debian, Red Hat, Mandrake e altri.

Senza un esplicito permesso scritto da parte dell’autore, è fatto divieto a distributori ed editori (compresi gli editori on-line) di imporre qualsivoglia condizione, restrizione o clausola aggiuntive al presente documento o a qualsiasi sua precedente versione. A partire dal presente aggiornamento, l’autore dichiara di *non* aver sottoscritto alcun impegno contrattuale che possa modificare le precedenti dichiarazioni.

In sostanza, il formato elettronico *inalterato* di questo libro può essere distribuito liberamente. Occorre ottenere il permesso dell’autore per la distribuzione di una sua versione modificata in modo sostanziale o di un’opera da esso derivata. Lo scopo di queste limitazioni è la preservazione dell’integrità artistica del documento ed evitare il “forking”.

Se si vuol esibire o distribuire questo documento, o qualsiasi precedente versione dello stesso, senza alcuna licenza se non quella di cui sopra, allora è necessario ottenere il permesso scritto dell’autore. La sua omissione può determinare la cessazione dei diritti di distribuzione.

Questi sono termini molto liberali e non dovrebbero ostacolare l’uso o la distribuzione legittima di questo libro. L’autore incoraggia, in particolare, il suo utilizzo per finalità scolastiche e di didattica.

Sono disponibili i diritti per la stampa commerciale ed altro. Se interessati, si è pregati di contattare l’autore (mailto:thegrendel@theriver.com).

L’autore ha prodotto questo libro coerentemente con lo spirito del Manifesto LDP (http://www.tldp.org/manifesto.html).

Hyun Jin Cha ha effettuato la traduzione in coreano (http://kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/index.html) della versione 1.0.11. Sono disponibili, o in corso, traduzioni in spagnolo, portoghese, francese, tedesco, italiano (http://it.tldp.org/guide/abs/index.html), russo (http://gazette.linux.ru.net/rus/articles/index-abs-guide.html) e cinese. Se si desidera tradurre questo documento in un'altra lingua, lo si può fare liberamente nei termini più sopra stabiliti. L'autore desidera essere informato di tali sforzi.