Corso facilitato di bash scripting

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

Parte 3

Nella quarta parte sono incluse interamente i contenuti riportati dall'originale.

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_SUBSHELL`

variabile che indica il livello della subshell. Si tratta di una nuova variabile aggiunta in Bash, versione 3.

Per il suo impiego vedi Esempio 20-1.

`$BASH_VERSINFO[n]`

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

```
  1 # Informazioni sulla versione Bash:
  2
  3 for n in 0 1 2 3 4 5
  4 do
  5   echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
  6 done
  7
  8 # BASH_VERSINFO[0] = 2                      # nr. della major
version.
  9 # BASH_VERSINFO[1] = 05                     # nr. della minor
version.
 10 # BASH_VERSINFO[2] = 8                      # nr. del patch level.
 11 # BASH_VERSINFO[3] = 1                      # nr. della build
version.
 12 # BASH_VERSINFO[4] = release                # Stato della release.
 13 # BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architettura.
 14                                             # (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 a qualsiasi identità l'utente corrente abbia assunto, solitamente tramite il comando su.

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

`$FUNCNAME`

nome della funzione corrente

```
  1 xyz23 ()
  2 {
  3   echo "$FUNCNAME è in esecuzione."  # xyz23 è in esecuzione.
  4 }
  5
  6 xyz23
  7
  8 echo "NOME FUNZIONE = $FUNCNAME"     # NOME FUNZIONE =
  9                                      # 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 Esempio 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 Esempio 9-13.

`$HOSTTYPE`

tipo di macchina

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

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

`$IFS`

separatore di campo (internal field separator)

Questa variabile determina il modo in cui Bash riconosce i campi, ovvero le singole parole, nell'interpratazione delle stringhe di caratteri.

Il valore preimpostato 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 Esempio 5-1.

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


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

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

Esempio 9-1. $IFS e gli spazi

```
  1 #!/bin/bash
  2 # $IFS gestisce gli spazi in modo diverso dagli altri
caratteri.
  3
  4 output_arg_uno_per_riga()
  5 {
  6   for arg
  7   do echo "[$arg]"
  8   done
  9 }
 10
 11 echo; echo "IFS=\" \""
 12 echo "-------"
 13
```

```
 14 IFS=" "
 15 var=" a  b c   "
 16 output_arg_uno_per_riga $var   # output_arg_uno_per_riga
`echo " a  b c   "`
 17 #
 18 # [a]
 19 # [b]
 20 # [c]
 21
 22
 23 echo; echo "IFS=:"
 24 echo "-----"
 25
 26 IFS=:
 27 var=":a::b:c:::"               # Come prima, ma con ":"
anziché " ".
 28 output_arg_uno_per_riga $var
 29 #
 30 # []
 31 # [a]
 32 # []
 33 # [b]
 34 # [c]
 35 # []
 36 # []
 37 # []
 38
 39 # In awk si ottiene lo stesso risultato con il separatore
di campo "FS".
 40
 41 # Grazie, Stephane Chazelas.
 42
 43 echo
 44
 45 exit 0
```

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

Vedi anche Esempio 12-34 per un'istruttiva dimostrazione sull'impiego di `$IFS`.

`$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.

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.

`$LINENO`

Variabile contenente 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.

```
  1 # *** INIZIO BLOCCO DI DEBUG ***
  2 ultimo_cmd_arg=$_  # Viene salvato.
  3
  4 echo "Alla riga numero $LINENO, variabile \"v1\" = $v1"
  5 echo "Ultimo argomento eseguito = $ultimo_cmd_arg"
  6 # *** FINE BLOCCO DI DEBUG ***
```

`$MACHTYPE`

tipo di macchina

Identifica il sistema hardware in modo dettagliato.

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

`$OLDPWD`

directory di lavoro precedente ("OLD-print-working-directory", la directory 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 Capitolo 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).

La "directory di lavoro" corrente `./`, di solito per ragioni di sicurezza, non è compresa in `$PATH`.

`$PIPESTATUS`

Array contenente lo/gli exit status dell'ultima pipe eseguita in *foreground* (primo piano). È piuttosto interessante in quanto non fornisce necessariamente come risultato l'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
```

Gli elemnti dell'array `$PIPESTATUS` sono gli exit status dei corrispettivi comandi eseguiti in una pipe. `$PIPESTATUS[0]` contiene l'exit status del primo comando della pipe, `$PIPESTATUS[1]` l'exit status del secondo comando, e così via.

La variabile `$PIPESTATUS`, in una shell di login, potrebbe contenere un errato valore 0 (nelle versioni Bash precedenti alla 3.0).

```
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.

Nella versione 2.05b.0(1)-release di Bash, e forse anche nelle versioni precedenti, la variabile `$PIPESTATUS` sembra non comportarsi correttamente. Si tratta di un errore che è stato (in gran parte) corretto nella versione 3.0 e seguente.

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

bash$ $ ls | comando_errato | wc
bash: comando_errato: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
141 127 0
```

```
bash$ echo $BASH_VERSION
3.00.0(1)-release

bash$ $ ls | comando_errato | wc
bash: comando_errato: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0
```

`$PIPESTATUS` è una variabile "volatile". Deve essere visualizzata immediatamente dopo la pipe, prima che venga eseguito qualsiasi altro comando.

```
bash$ $ ls | comando_errato | wc
bash: comando_errato: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0

bash$ echo ${PIPESTATUS[@]}
0
```

`$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 Esempio 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.

```
  1 #!/bin/bash
  2
  3 E_ERRATA_DIRECTORY=73
  4
  5 clear # Pulisce lo schermo.
  6
  7 DirectoryDestinazione=/home/bozo/projects/GreatAmericanNovel
  8
  9 cd $DirectoryDestinazione
 10 echo "Cancellazione dei vecchi file in $DirectoryDestinazione."
 11
 12 if [ "$PWD" != "$DirectoryDestinazione" ]
 13 then    # Evita di cancellare per errore una directory sbagliata.
 14   echo "Directory errata!"
 15   echo "Sei in $PWD, non in $DirectoryDestinazione!"
 16   echo "Salvo!"
 17   exit $E_ERRATA_DIRECTORY
 18 fi
 19
 20 rm -rf *
 21 rm .[A-Za-z0-9]*    # Cancella i file i cui nomi iniziano con un
punto.
 22 # rm -f .[^.]* ..?*   per cancellare file che iniziano con due o più
punti.
 23 # (shopt -s dotglob; rm -f *)   anche in questo modo.
 24 # Grazie, S.C. per la puntualizzazione.
 25
 26 #  I nomi dei file possono essere formati da tutti i caratteri
nell'intervallo
 27 #+ 0 - 255, tranne "/". La cancellazione di file che iniziano con
caratteri
 28 #+ inconsueti è lasciata come esercizio.
 29
 30 # Altre eventuali operazioni.
 31
 32 echo
 33 echo "Fatto."
 34 echo "Cancellati i vecchi file in $DirectoryDestinazione."
 35 echo
 36
 37
 38 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.

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

```
 3
 4 # REPLY è il valore preimpostato per il comando 'read'.
 5
 6 echo
 7 echo -n "Qual'è la tua verdura preferita?"
 8 read
 9
10 echo "La tua verdura preferita è $REPLY."
11 #  REPLY contiene il valore dell'ultimo  "read" se e solo se
12 #+ non è stata indicata alcuna variabile.
13
14 echo
15 echo -n "Qual'è il tuo frutto preferito?"
16 read frutto
17 echo "Il tuo frutto preferito è $frutto."
18 echo "ma..."
19 echo "Il valore di \$REPLY è ancora $REPLY."
20 #  $REPLY è ancora impostato al valore precedente perché
21 #+ la variabile $frutto contiene il nuovo valore letto con "read".
22
23 echo
24
25 exit 0
```

`$SECONDS`

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

```
 1 #!/bin/bash
 2
 3 TEMPO_LIMITE=10
 4 INTERVALLO=1
 5
 6 echo
 7 echo "Premi Control-C per terminare prima di $TEMPO_LIMITE secondi."
 8 echo
 9
10 while [ "$SECONDS" -le "$TEMPO_LIMITE" ]
11 do
12   if [ "$SECONDS" -eq 1 ]
13   then
14     unita=secondo
15   else
16     unita=secondi
17   fi
18
19   echo "Questo script è in esecuzione da $SECONDS $unita."
20   #  Su una macchina lenta o sovraccarica lo script, talvolta,
21   #+ potrebbe saltare un conteggio.
22   sleep $INTERVALLO
23 done
24
25 echo -e "\a"  # Beep!
26
27 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.

```
 1 # Funziona negli script con Bash versione 2.05b e successive.
 2
 3 TMOUT=3    # Imposta il prompt alla durata di tre secondi.
 4
 5 echo "Qual'è la tua canzone preferita?"
 6 echo "Svelto, hai solo $TMOUT secondi per rispondere!"
 7 read canzone
 8
 9 if [ -z "$canzone" ]
10 then
11   canzone="(nessuna risposta)"
12   # Risposta preimpostata.
13 fi
14
15 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 Esempio 30-5) l'interrupt generato dal ciclo di temporizzazione (fiu!).

Esempio 9-2. Input temporizzato

```
 1 #!/bin/bash
 2 # timed-input.sh
 3
 4 # TMOUT=3      Funziona anche questo, a partire dalle più recenti
 5 #              versioni di Bash.
 6
 7 TEMPOLIMITE=3  #  In questo caso tre secondi. Può essere impostato
 8                #+ ad un valore diverso.
 9
10 VisualizzaRisposta()
11 {
12   if [ "$risposta" = TIMEOUT ]
13   then
14     echo $risposta
15   else       # Ho voluto tenere separati i due esempi.
16     echo "La tua verdura preferita è $risposta"
17     kill $!  #  Uccide la funzione AvvioTimer in esecuzione in
18              #+ background perché non più necessaria. $! è il PID
```

```
 19              #+ dell'ultimo job in esecuzione in background.
 20
 21   fi
 22
 23 }
 24
 25
 26
 27 AvvioTimer()
 28 {
 29   sleep $TEMPOLIMITE && kill -s 14 $$ &
 30   # Attende 3 secondi, quindi invia il segnale SIGALARM allo script.
 31 }
 32
 33 Int14Vettore()
 34 {
 35   risposta="TIMEOUT"
 36   VisualizzaRisposta
 37   exit 14
 38 }
 39
 40 trap Int14Vettore 14   # Interrupt del timer (14) modificato allo
scopo.
 41
 42 echo "Qual'è la tua verdura preferita? "
 43 AvvioTimer
 44 read risposta
 45 VisualizzaRisposta
 46
 47 #  Ammettiamolo, questa è un'implementazione tortuosa per
temporizzare
 48 #+ l'input, comunque l'opzione "-t" di "read" semplifica il compito.
 49 #  Vedi sopra "t-out.sh".
 50
 51 #  Se desiderate qualcosa di più elegante... prendete in
considerazione
 52 #+ la possibilità di scrivere l'applicazione in C o C++,
 53 #+ utilizzando le funzioni di libreria appropriate, come 'alarm' e
'setitimer'.
 54
 55 exit 0
```

Un'alternativa è l'utilizzo di stty.

Esempio 9-3. Input temporizzato, un ulteriore esempio

```
  1 #!/bin/bash
  2 # timeout.sh
  3
  4 #  Scritto da Stephane Chazelas
  5 #+ e modificato dall'autore del libro.
  6
  7 INTERVALLO=5              # intervallo di timeout
  8
  9 leggi_temporizzazione() {
 10   timeout=$1
 11   nomevar=$2
 12   precedenti_impostazioni_tty=`stty -g`
 13   stty -icanon min 0 time ${timeout}0
 14   eval read $nomevar      # o semplicemente  read $nomevar
 15   stty "$precedenti_impostazioni_tty"
```

```
 16   # Vedi la pagina di manuale di "stty".
 17 }
 18
 19 echo; echo -n "Come ti chiami? Presto! "
 20 leggi_temporizzazione $INTERVALLO nome
 21
 22 #  Questo potrebbe non funzionare su tutti i tipi di terminale.
 23 #  Il timeout massimo, infatti, dipende dallo specifico terminale.
 24 #+ (spesso è di 25.5 secondi).
 25
 26 echo
 27
 28 if [ ! -z "$nome" ]  # se il nome è stato immesso prima del
timeout...
 29 then
 30   echo "Ti chiami $nome."
 31 else
 32   echo "Tempo scaduto."
 33 fi
 34
 35 echo
 36
 37 # Il comportamento di questo script è un po' diverso da "timed-
input.sh".
 38 # Ad ogni pressione di tasto, la temporizzazione ricomincia da capo.
 39
 40 exit 0
```

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

Esempio 9-4. read temporizzato

```
  1 #!/bin/bash
  2 # t-out.sh
  3 # Ispirato da un suggerimento di "syngin seven" (grazie).
  4
  5 TEMPOLIMITE=4        # 4 secondi
  6
  7
  8 read -t $TEMPOLIMITE variabile <&1
  9
 10 echo
 11
 12 if [ -z "$variabile" ]
 13 then
 14   echo "Tempo scaduto, la variabile non è stata impostata."
 15 else
 16   echo "variabile = $variabile"
 17 fi
 18
 19 exit 0
 20
 21 # Esercizio per il lettore:
 22 # ------------------------
 23 # Perché è necessaria la redirezione (<&1) alla riga 8?
 24 # 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.

Esempio 9-5. Sono root?

```
 1 #!/bin/bash
 2 # am-i-root.sh:   Sono root o no?
 3
 4 ROOT_UID=0   # Root ha $UID 0.
 5
 6 if [ "$UID" -eq "$ROOT_UID" ]  #  Il vero "root" avrà la compiacenza
 7                                #+ di aspettare?
 8 then
 9   echo "Sei root."
10 else
11   echo "Sei un utente normale (ma la mamma ti vuol bene lo stesso)."
12 fi
13
14 exit 0
15
16
17 #
=================================================================== #
18 # Il codice seguente non viene eseguito perché lo script è già
terminato.
19
20 # Un metodo alternativo per andare al fondo della questione:
21
22 NOME_ROOT=root
23
24 nomeutente=`id -nu`              # Oppure...   nomeutente=`whoami`
25 if [ "$nomeutente" = "$NOME_ROOT" ]
26 then
27   echo "Rooty, toot, toot. Sei root."
28 else
29   echo "Sei solo un semplice utente."
30 fi
```

Vedi anche Esempio 2-3.

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 Esempio 4-5 e Esempio 11-14)

`$#`

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

`$*`

Tutti i parametri posizionali visti come un'unica parola

 "`$*`" 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.

 Naturalmente, "`$@`" va usata con il quoting.

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

```
 1 #!/bin/bash
 2 # arglist.sh
 3 # Invocate lo script con molti argomenti, come "uno due tre".
 4
 5 E_ERR_ARG=65
 6
 7 if [ ! -n "$1" ]
 8 then
 9   echo "Utilizzo: `basename $0` argomento1 argomento2 ecc."
10   exit $E_ERR_ARG
11 fi
12
13 echo
14
15 indice=1         # Inizializza il contatore.
16
17 echo "Elenco degli argomenti con \"\$*\":"
18 for arg in "$*"  # Non funziona correttamente se "$*" non è tra
apici.
19 do
```

```
 20   echo "Argomento nr.$indice = $arg"
 21   let "indice+=1"
 22 done             # $* vede tutti gli argomenti come un'unica parola.
 23 echo "Tutto l'elenco come parola singola."
 24
 25 echo
 26
 27 indice=1         # Reimposta il contatore.
 28                  # Cosa succede se vi dimenticate di farlo?
 29
 30 echo "Elenco degli argomenti con \"\$@\":"
 31 for arg in "$@"
 32 do
 33   echo "Argomento nr.$indice = $arg"
 34   let "indice+=1"
 35 done             # $@ vede gli argomenti come parole separate.
 36 echo "Elenco composto da diverse parole."
 37
 38 echo
 39
 40 indice=1         # Reimposta il contatore.
 41
 42 echo "Eleco degli argomenti con \$* (senza quoting):"
 43 for arg in $*
 44 do
 45   echo "Argomento nr.$indice = $arg"
 46   let "indice+=1"
 47 done             # $* senza quoting vede gli argomenti come parole
separate.
 48 echo "Elenco composto da diverse parole."
 49
 50 exit 0
```

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

```
  1 #!/bin/bash
  2 # Da eseguire con  ./nomescript 1 2 3 4 5
  3
  4 echo "$@"    # 1 2 3 4 5
  5 shift
  6 echo "$@"    # 2 3 4 5
  7 shift
  8 echo "$@"    # 3 4 5
  9
 10 # Ogni "shift" perde il precedente $1.
 11 # Come conseguenza "$@" 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 Esempio 12-20 e Esempio 12-21.


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

Esempio 9-7. Comportamento incoerente di `$*` e `$@`

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

```
  2
  3 #  Comportamento non corretto delle variabili interne Bash "$*" e
"$@",
  4 #+ dipendente dal fatto che vengano utilizzate o meno con il
"quoting".
  5 #  Gestione incoerente della suddivisione delle parole e del ritorno
a capo.
  6
  7 set -- "Il primo" "secondo" "il:terzo" "" "Il: :quinto"
  8 # Imposta gli argomenti dello script, $1, $2, ecc.
  9
 10 echo
 11
 12 echo 'IFS con il valore preimpostato, utilizzando "$*"'
 13 c=0
 14 for i in "$*"               # tra doppi apici
 15 do echo "$((c+=1)): [$i]"   # Questa riga rimane invariata in tutti
gli esempi.
 16                             # Visualizza gli argomenti.
 17 done
 18 echo ---
 19
 20 echo 'IFS con il valore preimpostato, utilizzando $*'
 21 c=0
 22 for i in $*                 # senza apici
 23 do echo "$((c+=1)): [$i]"
 24 done
 25 echo ---
 26
 27 echo 'IFS con il valore preimpostato, utilizzando "$@"'
 28 c=0
 29 for i in "$@"
 30 do echo "$((c+=1)): [$i]"
 31 done
 32 echo ---
 33
 34 echo 'IFS con il valore preimpostato, utilizzando $@'
 35 c=0
 36 for i in $@
 37 do echo "$((c+=1)): [$i]"
 38 done
 39 echo ---
 40
 41 IFS=:
 42 echo 'IFS=":", utilizzando "$*"'
 43 c=0
 44 for i in "$*"
 45 do echo "$((c+=1)): [$i]"
 46 done
 47 echo ---
 48
 49 echo 'IFS=":", utilizzando $*'
 50 c=0
 51 for i in $*
 52 do echo "$((c+=1)): [$i]"
 53 done
 54 echo ---
 55
 56 var=$*
 57 echo 'IFS=":", utilizzando "$var" (var=$*)'
 58 c=0
 59 for i in "$var"
```

```
 60 do echo "$((c+=1)): [$i]"
 61 done
 62 echo ---
 63
 64 echo 'IFS=":", utilizzando $var (var=$*)'
 65 c=0
 66 for i in $var
 67 do echo "$((c+=1)): [$i]"
 68 done
 69 echo ---
 70
 71 var="$*"
 72 echo 'IFS=":", utilizzando $var (var="$*")'
 73 c=0
 74 for i in $var
 75 do echo "$((c+=1)): [$i]"
 76 done
 77 echo ---
 78
 79 echo 'IFS=":", utilizzando "$var" (var="$*")'
 80 c=0
 81 for i in "$var"
 82 do echo "$((c+=1)): [$i]"
 83 done
 84 echo ---
 85
 86 echo 'IFS=":", utilizzando "$@"'
 87 c=0
 88 for i in "$@"
 89 do echo "$((c+=1)): [$i]"
 90 done
 91 echo ---
 92
 93 echo 'IFS=":", utilizzando $@'
 94 c=0
 95 for i in $@
 96 do echo "$((c+=1)): [$i]"
 97 done
 98 echo ---
 99
100 var=$@
101 echo 'IFS=":", utilizzando $var (var=$@)'
102 c=0
103 for i in $var
104 do echo "$((c+=1)): [$i]"
105 done
106 echo ---
107
108 echo 'IFS=":", utilizzando "$var" (var=$@)'
109 c=0
110 for i in "$var"
111 do echo "$((c+=1)): [$i]"
112 done
113 echo ---
114
115 var="$@"
116 echo 'IFS=":", utilizzando "$var" (var="$@")'
117 c=0
118 for i in "$var"
119 do echo "$((c+=1)): [$i]"
120 done
121 echo ---
```

```
122
123 echo 'IFS=":", utilizzando $var (var="$@")'
124 c=0
125 for i in $var
126 do echo "$((c+=1)): [$i]"
127 done
128
129 echo
130
131 # Provate questo script con ksh o zsh -y.
132
133 exit 0
134
135 # Script d'esempio di Stephane Chazelas,
136 # con piccole modifiche apportate dall'autore.
```

 I parametri **$@** e **$*** differiscono solo quando vengono posti tra doppi apici.

Esempio 9-8. `$*` e `$@` quando `$IFS` è vuoto

```
 1 #!/bin/bash
 2
 3 #  Se $IFS è impostata, ma vuota, allora "$*" e "$@" non
 4 #+ visualizzano i parametri posizionali come ci si aspetterebbe.
 5
 6 mecho ()       # Visualizza i parametri posizionali.
 7 {
 8 echo "$1,$2,$3";
 9 }
10
11
12 IFS=""         # Impostata, ma vuota.
13 set a b c      # Parametri posizionali.
14
15 mecho "$*"     # abc,,
16 mecho $*       # a,b,c
17
18 mecho $@       # a,b,c
19 mecho "$@"     # a,b,c
20
21 #  Il comportamento di $* e $@ quando $IFS è vuota dipende da quale
22 #+ versione Bash o sh è in esecuzione. È quindi sconsigliabile fare
23 #+ affidamento su questa "funzionalità" in uno script.
24
25
26 # Grazie, S.C.
27
28 exit 0
```

Altri Parametri Speciali

`$-`

Opzioni passate allo script (utilizzando set). Vedi Esempio 11-14.

 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

```
  1 LOG=$0.log
  2
  3 COMANDO1="sleep 100"
  4
  5 echo "Registra i PID dei comandi in background dello script: $0" >>
"$LOG"
  6 # Possono essere così controllati e, se necessario, "uccisi".
  7 echo >> "$LOG"
  8
  9 # Registrazione dei comandi.
 10
 11 echo -n "PID di \"$COMANDO1\":  " >> "$LOG"
 12 ${COMANDO1} &
 13 echo $! >> "$LOG"
 14 # PID di "sleep 100":  1506
 15
 16 # Grazie a Jacques Lederer, per il suggerimento.
```

```
  1 possibile_job_bloccato & { sleep ${TIMEOUT}; eval 'kill -9 $!' &>
/dev/null; }
  2 # Forza il completamento di un programma mal funzionante.
  3 # Utile, ad esempio, negli script init.
  4
  5 # Grazie a Sylvain Fourmanoit per aver segnalato quest'uso creativo
della variabile "!".
```

$_

Variabile speciale impostata all'ultimo argomento del precedente comando eseguito.

Esempio 9-9. Variabile underscore

```
  1 #!/bin/bash
  2
  3 echo $_              # /bin/bash
  4                      # digitate solo /bin/bash per eseguire lo
script.
  5
  6 du >/dev/null        # Non viene visualizzato alcun output del
comando.
  7 echo $_              # du
  8
  9 ls -al >/dev/null    # Non viene visualizzato alcun output del
comando.
 10 echo $_              # -al  (ultimo argomento)
 11
 12 :
 13 echo $_              # :
```

$?

Exit status di un comando, funzione, o dello stesso script (vedi Esempio 23-6)

$$

ID di processo dello script. La variabile $$ viene spesso usata negli script per creare un nome di file temporaneo "unico" (vedi Esempio A-14, Esempio 30-6, Esempio 12-26 e Esempio 11-24). Di solito è più semplice che invocare mktemp.

Note

[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.

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" : '.*'

```
1 stringaZ=abcABC123ABCabc
2
3 echo ${#stringaZ}                 # 15
4 echo `expr length $stringaZ`      # 15
5 echo `expr "$stringaZ" : '.*'`    # 15
```

Esempio 9-10. Inserire una riga bianca tra i paragrafi di un file di testo

```
 1 #!/bin/bash
 2 # paragraph-space.sh
 3
 4 #  Inserisce una riga bianca tra i paragrafi di un file di testo con
 5 #+ spaziatura semplice.
 6 #  Utilizzo: $0 <NOMEFILE
 7
 8 LUNMIN=45        # Potrebbe rendersi necessario modificare questo valore.
 9 #  Si assume che le righe di lunghezza inferiore a $LUNMIN caratteri
10 #+ siano le ultime dei paragrafi.
11
12 while read riga  # Per tutte le righe del file di input...
13 do
14   echo "$riga"   # Visualizza la riga.
15
16   len=${#riga}
17   if [ "$len" -lt "$LUNMIN" ]
18     then echo    # Aggiunge la riga bianca.
19   fi
20 done
21
22 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.

```
 1 stringaZ=abcABC123ABCabc
 2 #        |------|
 3
 4 echo `expr match "$stringaZ" 'abc[A-Z]*.2'`   # 8
 5 echo `expr "$stringaZ" : 'abc[A-Z]*.2'`       # 8
```

Indice

expr index $stringa $sottostringa

Numero di posizione in $stringa del primo carattere presente in $sottostringa che è stato verificato.

```
 1 stringaZ=abcABC123ABCabc
 2 echo `expr index "$stringaZ" C12`             # 6
 3                                               # Posizione di C.
 4
 5 echo `expr index "$stringaZ" 1c`              # 3
 6 # '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 `$stringa` è "*" o "@", allora vengono estratti i parametri posizionali, [1] iniziando da `$posizione`.

${stringa:posizione:lunghezza}

Estrae una sottostringa di *`$lunghezza`* caratteri da *`$stringa`* iniziando da *`$posizione`*.

```
 1 stringaZ=abcABC123ABCabc
 2 #        0123456789.....
 3 #        L'indicizzazione inizia da 0.
 4
 5 echo ${stringaZ:0}                            # abcABC123ABCabc
 6 echo ${stringaZ:1}                            # bcABC123ABCabc
 7 echo ${stringaZ:7}                            # 23ABCabc
 8
 9 echo ${stringaZ:7:3}                          # 23A
10                                               # Sottostringa di tre
caratteri.
```

```
11
12
13
14 # È possibile indicizzare partendo dalla fine della stringa?
15
16 echo ${stringaZ:-4}                           # abcABC123ABCabc
17 # Restituisce l'intera stringa, come con ${parametro:-default}.
18 # Tuttavia . . .
19
20 echo ${stringaZ:(-4)}                         # Cabc
21 echo ${stringaZ: -4}                          # Cabc
22 # Ora funziona.
23 # Le parentesi, o l'aggiunta di uno spazio, "preservano" il
parametro negativo.
24
25 # Grazie, Dan Jacobson, per averlo evidenziato.
```

Se il parametro `$stringa` è "*" o "@", vengono estratti un massimo di `$lunghezza` parametri posizionali, iniziando da `$posizione`.

```
 1 echo ${*:2}          #  Visualizza tutti i parametri iniziando dal
secondo.
 2 echo ${@:2}          #  Come prima.
 3
 4 echo ${*:2:3}        #  Visualizza tre parametri posizionali
 5                      #+ iniziando dal secondo.
```

expr substr $stringa $posizione $lunghezza

Estrae *`$lunghezza`* caratteri da *`$stringa`* iniziando da *`$posizione`*.

```
 1 stringaZ=abcABC123ABCabc
 2 #        123456789......
 3 #        L'indicizzazione inizia da 1.
 4
 5 echo `expr substr $stringaZ 1 2`              # ab
 6 echo `expr substr $stringaZ 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.

```
 1 stringaZ=abcABC123ABCabc
 2 #        =======
 3
 4 echo `expr match "$stringaZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
 5 echo `expr "$stringaZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
 6 echo `expr "$stringaZ" : '\(.......\)'`                   # abcABC1
 7 # 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.

```
  1 stringaZ=abcABC123ABCabc
  2 #                ======
  3
  4 echo `expr match "$stringaZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    #
ABCabc
  5 echo `expr "$stringaZ" : '.*\(......\)'`                       #
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*.

```
  1 stringaZ=abcABC123ABCabc
  2 #        |----|
  3 #        |----------|
  4
  5 echo ${stringaZ#a*C}      # 123ABCabc
  6 # È stata tolta l'occorrenza più breve compresa tra 'a' e 'C'.
  7
  8 echo ${stringaZ##a*C}     # abc
  9 # 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*.

```
  1 stringaZ=abcABC123ABCabc
  2 #                     ||
  3 #         |------------|
  4
  5 echo ${stringaZ%b*c}      # abcABC123ABCa
  6 #  È stata tolta l'occorrenza più breve compresa
  7 #+ tra 'b' e 'c', dalla fine di $stringaZ.
  8
  9 echo ${stringaZ%%b*c}     # a
 10 #  È stata tolta l'occorrenza più lunga compresa
 11 #+ tra 'b' e 'c', dalla fine di $stringaZ.
```

Esempio 9-11. Conversione di formato di file grafici e modifica del nome dei file

```
  1 #!/bin/bash
  2 #  cvt.sh:
  3 #  Converte tutti i file immagine MacPaint, in una directory data,
  4 #+ nel formato "pbm".
  5 #  Viene utilizzato l'eseguibile "macptopbm" del pacchetto "netpbm",
  6 #+ mantenuto da Brian Henderson (bryanh@giraffe-data.com). Netpbm di
  7 #+ solito è compreso nella maggior parte delle distribuzioni
standard Linux.
  8
  9 OPERAZIONE=macptopbm
 10 ESTENSIONE=pbm          # Nuova estensione dei nomi dei file.
 11
 12 if [ -n "$1" ]
 13 then
 14   directory=$1      #  Se viene fornito il nome di una directory
come
 15                     #+ argomento dello script...
 16 else
 17   directory=$PWD    # Altrimenti viene utilizzata la directory
corrente.
 18 fi
 19
 20 #  Si assume che tutti i file immagine nella directory siano dei
MacPaint,
 21 # + con estensione ".mac".
 22
 23 for file in $directory/*    # Globbing dei nomi dei file.
 24 do
 25   nomefile=${file%.*c}      #  Toglie l'estensione ".mac" dal nome
del file
 26                             #+ ('.*c' verifica tutto tra '.' e 'c',
compresi).
 27   $OPERAZIONE $file > "$nomefile.$ESTENSIONE"
 28                             #  Converte e redirige il file con una
nuova
 29                             #+ estensione.
 30   rm -f $file               #  Cancella i file originali dopo la
conversione.
 31   echo "$nomefile.$ESTENSIONE"  # Visualizza quello che avviene allo
stdout.
 32 done
 33
 34 exit 0
 35
 36 # Esercizio:
 37 # ----------
 38 #  Così com'è, lo script converte "tutti" i file presenti nella
 39 #+ directory di lavoro corrente.
 40 #  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*.

```
  1 stringaZ=abcABC123ABCabc
  2
  3 echo ${stringaZ/abc/xyz}   # xyzABC123ABCabc
  4                            # Sostituisce la prima occorrenza di
'abc' con 'xyz'.
  5
  6 echo ${stringaZ//abc/xyz}  # xyzABC123ABCxyz
  7                            # 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*.

```
  1 stringaZ=abcABC123ABCabc
  2
  3 echo ${stringaZ/#abc/XYZ}     # XYZABC123ABCabc
  4                               # Sostituisce l'occorrenza iniziale
'abc'con'XYZ'.
  5
  6 echo ${stringaZ/%abc/XYZ}     # abcABC123ABCXYZ
  7                               # 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.

Esempio 9-12. Modi alternativi di estrarre sottostringhe

```
 1 #!/bin/bash
 2 # substring-extraction.sh
 3
 4 Stringa=23skidoo1
 5 #       012345678    Bash
 6 #       123456789    awk
 7 # Fate attenzione al diverso sistema di indicizzazione della stringa:
 8 # Bash numera il primo carattere della stringa con '0'.
 9 # Awk  numera il primo carattere della stringa con '1'.
10
11 echo ${Stringa:2:4} # posizione 3 (0-1-2), 4 caratteri di lunghezza
12                                        # skid
13
14 #  L'equivalente awk di ${stringa:pos:lunghezza} è
15 #+ substr(stringa,pos,lunghezza).
16 echo | awk '
17 { print substr("'"${Stringa}"'",3,4)     # skid
18 }
19 '
```

```
20 #  Collegando ad awk un semplice comando "echo" gli viene dato un
21 #+ input posticcio, in questo modo non diventa più necessario
22 #+ fornirgli il nome di un file.
23
24 exit 0
```

9.2.2. Ulteriori approfondimenti

Per altro materiale sulla manipolazione delle stringhe negli script, si faccia riferimento alla Sezione 9.3 e all' importante sezione relativa all'elenco dei comandi expr. Per gli script d'esempio, si veda:

1. Esempio 12-9
2. Esempio 9-15
3. Esempio 9-16
4. Esempio 9-17
5. Esempio 9-19

Note

[1] Questo vale sia per gli argomenti da riga di comando che per i parametri passati ad una funzione

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.

```
  1 tuo_id=${USER}-su-${HOSTNAME}
  2 echo "$tuo_id"
  3 #
  4 echo "Vecchio \$PATH = $PATH"
  5 PATH=${PATH}:/opt/bin  # Aggiunge /opt/bin a $PATH per la durata
dello script.
  6 echo "Nuovo \$PATH = $PATH"
```

`${parametro-default}`, **`${parametro:-default}`**

Se parametro non è impostato, viene impostato al valore fornito da default.

```
  1 echo ${nomeutente-`whoami`}
  2 #  Visualizza il risultato del comando `whoami`, se la variabile
  3 #+ $nomeutente non è ancora impostata.
```

`${parametro-default}` e *`${parametro:-default}`* sono quasi uguali. L'aggiunta dei : serve solo quando *parametro* è stato dichiarato, ma non impostato.

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

```
  2 # param-sub.sh
  3
  4 #  Il fatto che una vairabile sia stata dichiarata
  5 #+ influenza l'uso dell'opzione preimpostata,
  6 #+ anche se la variabile è nulla.
  7
  8 nomeutente0=
  9 # nomeutente0 è stata dichiarata, ma contiene un valore nullo.
 10 echo "nomeutente0 = ${nomeutente0-`whoami`}"
 11 # Non visualizza niente.
 12
 13 echo "nomeutente1 = ${nomeutente1-`whoami`}"
 14 # nomeutente1 non è stata dichiarata.
 15 # Viene visualizzato.
 16
 17 nomeutente2=
 18 # nomeutente2 è stata dichiarata, ma contiene un valore nullo.
 19 echo "nomeutente2 = ${nomeutente2:-`whoami`}"
 20 # Viene visualizzato perché sono stati utilizzati :- al posto del
semplice -.
 21 # Confrontatelo con il primo esempio visto sopra.
 22
 23 exit 0
```

Il costrutto *parametro-default* viene utilizzato per fornire agli script gli argomenti "dimenticati" da riga di comando.

```
  1 DEFAULT_NOMEFILE=generico.dat
  2 nomefile=${1:-$DEFAULT_NOMEFILE}
  3 #  Se non diversamente specificato, il successivo blocco di
  4 #+ comandi agisce sul file "generico.dat".
  5 #
  6 #  Seguono comandi.
```

Vedi anche Esempio 3-4, Esempio 29-2 e Esempio 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, [1] come visto in precedenza.

```
  1 echo ${nomeutente=`whoami`}
  2 # 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.

```
 1 echo "###### \${parametro+altro_valore} ########"
 2 echo
 3
 4 a=${param1+xyz}
 5 echo "a = $a"      # a =
 6
 7 param2=
 8 a=${param2+xyz}
 9 echo "a = $a"      # a = xyz
10
11 param3=123
12 a=${param3+xyz}
13 echo "a = $a"      # a = xyz
14
15 echo
16 echo "###### \${parametro:+altro_valore} ########"
17 echo
18
19 a=${param4:+xyz}
20 echo "a = $a"      # a =
21
22 param5=
23 a=${param5:+xyz}
24 echo "a = $a"      # a =
25 # Risultato diverso da   a=${param5+xyz}
26
27 param6=123
28 a=${param6+xyz}
29 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.

Esempio 9-13. Sostituzione di parametro e messaggi d'errore

```
 1 #!/bin/bash
 2
 3 #  Verifica alcune delle variabili d'ambiente di sistema.
 4 #  È una buona misura preventiva.
 5 #  Se, per sempio, $USER, il nome dell'utente corrente, non è impostata,
 6 #+ la macchina non può riconoscervi.
 7
 8 : ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
 9   echo
10   echo "Il nome della macchina è $HOSTNAME."
11   echo "Tu sei $USER."
12   echo "La directory home è $HOME."
13   echo "La cartella di posta INBOX si trova in $MAIL."
14   echo
15   echo "Se leggete questo messaggio, vuol dire che"
16   echo "le variabili d'ambiente più importanti sono impostate."
17   echo
18   echo
19
20 # ------------------------------------------------------
21
22 #  Il costrutto ${nomevariabile?} può verificare anche
```

```
 23 #+ le variabili impostate in uno script.
 24
 25 QuestaVariabile=Valore-di-Questa-Variabile
 26 #  È da notare, en passant, che le variabili stringa possono contenere
 27 #+ caratteri che non sono consentiti se usati nei loro nomi .
 28 : ${QuestaVariabile?}
 29 echo "Il valore di QuestaVariabile è $QuestaVariabile".
 30 echo
 31 echo
 32
 33
 34 : ${ZZXy23AB?"ZZXy23AB non è stata impostata."}
 35 #  Se ZZXy23AB non è stata impostata,
 36 #+ allora lo script termina con un messaggio d'errore.
 37
 38 # Il messaggio d'errore può essere specificato.
 39 # : ${ZZXy23AB?"ZZXy23AB non è stata impostata."}
 40
 41
 42 #  Stesso risultato con:
 43 #+     finta_variabile=${ZZXy23AB?}
 44 #+     finta_variabile=${ZZXy23AB?"ZXy23AB non è stata impostata."}
 45 #
 46 #      echo ${ZZXy23AB?} >/dev/null
 47
 48 #  Confrontate questi metodi per la verifica dell'impostazione di una
variabile
 49 #+ con "set -u" . . .
 50
 51
 52
 53 echo "Questo messaggio non viene visualizzato perché lo script è già
terminato."
 54
 55 QUI=0
 56 exit $QUI   # NON termina in questo punto.
```

Esempio 9-14. Sostituzione di parametro e messaggi "utilizzo"

```
  1 #!/bin/bash
  2 # usage-message.sh
  3
  4 : ${1?"Utilizzo: $0 ARGOMENTO"}
  5 #  Lo script termina qui, se non vi è un parametro da riga di comando,
  6 #+ e viene visualizzato il seguente messaggio d'errore.
  7 #    usage-message.sh: 1: Utilizzo: usage-message.sh ARGOMENTO
  8
  9 echo "Queste due righe vengono visualizzate solo se è stato
 10 fornito un argomento."
 11 echo "argomento da riga di comando = \"$1\""
 12
 13 exit 0  #  Lo script termina a questo punto solo se è stato
 14         #+ eseguito con l'argomento richiesto.
 15
 16 # Verificate l'exit status dello script eseguito, sia con che senza
argomento.
 17 # Se il parametro è stato fornito, allora  "$?" è  0.
 18 # Altrimenti "$?" è 1.
```

Sostituzione e/o espansione di parametro. Le espressioni che seguono sono il complemento delle operazioni sulle stringhe del costrutto **match** con **expr** (vedi Esempio 12-9). 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.

Eccezioni:

- **${#*}** e **${#@}** forniscono il *numero dei parametri posizionali*.
- Per gli array, **${#array[*]}** e **${#array[@]}** forniscono il numero degli elementi che compongono l'array.

Esempio 9-15. Lunghezza di una variabile

```
  1 #!/bin/bash
  2 # length.sh
  3
  4 E_NO_ARG=65
  5
  6 if [ $# -eq 0 ]  # Devono essere forniti degli argomenti allo
script.
  7 then
  8   echo "Eseguite lo script con uno o più argomenti."
  9   exit $E_NO_ARG
 10 fi
 11
 12 var01=abcdEFGH28ij
 13
 14 echo "var01 = ${var01}"
 15 echo "Lunghezza di var01 = ${#var01}"
 16
 17 echo "Numero di argomenti passati allo script = ${#@}"
 18 echo "Numero di argomenti passati allo script = ${#*}"
 19
 20 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'Esempio A-8:

```
  1 # Funzione dall'esempio "days-between.sh".
  2 # Toglie lo/gli zeri iniziali dall'argomento fornito.
  3
  4 toglie_zero_iniziale () #  Toglie possibili zeri iniziali
  5 {                       #+ dagli argomenti passati.
  6   return=${1#0}         #  "1" stà per $1 -- l'argomento passato.
  7 }                       #  "0" indica ciò che va tolto da "$1" --
gli zeri.
```

Variante, più elaborata dell'esempio precedente, di Manfred Schwarb:

```
  1 toglie_zero_iniziale2 () #  Toglie possibili zeri iniziali,
altrimenti
  2 {                        #+ Bash interpreta tali numeri come valori
ottali.
  3   shopt -s extglob       #  Abilita il globbing esteso.
  4   local val=${1##+(0)}   #  Usa una variabile locale, verifica
d'occorrenza più
  5                          #+ lunga delle serie di 0.
  6   shopt -u extglob       #  Disabilita il globbing esteso.
  7   _toglie_zero_iniziale2=${val:-0}
  8                          #  Nel caso l'input sia 0, restituisce 0
invece di "".
  9 }
```

Altro esempio di utilizzo:

```
  1 echo `basename $PWD`        # Nome della directory di lavoro
corrente.
  2 echo "${PWD##*/}"           # Nome della directory di lavoro
corrente.
  3 echo
  4 echo `basename $0`          # Nome dello script.
  5 echo $0                     # Nome dello script.
  6 echo "${0##*/}"            # Nome dello script.
  7 echo
  8 nomefile=test.dat
  9 echo "${nomefile##*.}"      # dat
 10                             # 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 ha introdotto delle opzioni aggiuntive.

Esempio 9-16. Ricerca di corrispondenza nella sostituzione di parametro

```
  1 #!/bin/bash
  2 # patt-matching.sh
  3
  4 #  Ricerca di corrispondenza utilizzando gli operatori si sostituzione
  5 #+ di parametro # ##  % %%.
  6
  7 var1=abcd12345abc6789
  8 modello1=a*c  #  * (carattere jolly) verifica tutto quello che
  9              #+ è compreso tra a - c.
 10
 11 echo
 12 echo "var1 = $var1"           # abcd12345abc6789
 13 echo "var1 = ${var1}"         # abcd12345abc6789
 14                               # (forma alternativa)
 15 echo "Numero di caratteri in ${var1} = ${#var1}"
 16 echo
 17
 18 echo "modello1 = $modello1"   # a*c  (tutto ciò che è compreso tra 'a' e
'c')
 19 echo "--------------"
```

```
20 echo '${var1#$modello1}  =' "${var1#$modello1}"    # d12345abc6789
21 #  All'eventuale occorrenza più corta, toglie i primi 3 caratteri
22 #+ abcd12345abc6789                               ^^^^^
23 #  |-|
24 echo '${var1##$modello1} =' "${var1##$modello1}"   # 6789
25 #  All'eventuale occorrenza più lunga, toglie i primi 12 caratteri
26 #+ abcd92345abc6789                               ^^^^^
27 #+ |----------|
28
29 echo; echo; echo
30
31 modello2=b*9            # tutto quello che si trova tra 'b' e '9'
32 echo "var1 = $var1"     # Ancora  abcd12345abc6789
33 echo
34 echo "modello2 = $modello2"
35 echo "--------------"
36
37 echo '${var1%modello2}  =' "${var1%$modello2}"     #     abcd12345a
38 #  All'eventuale occorrenza più corta, toglie gli ultimi 6 caratteri
39 #+ abcd12345abc6789                                 ^^^^^^
40 #+           |----|
41 echo '${var1%%modello2} =' "${var1%%$modello2}"    #     a
42 #  All'eventuale occorrenza più lunga, toglie gli ultimi 15 caratteri
43 #+ abcd12345abc6789                                 ^^^^^^
44 #+  |-------------|
45
46 #  Ricordate, # e ## agiscono sulla parte iniziale della stringa
47 #+            (da sinistra verso destra), % e %% agiscono sulla parte
48 #+            finale della stringa (da destra verso sinistra).
49
50 echo
51
52 exit 0
```

Esempio 9-17. Rinominare le estensioni dei file:

```
 1 #!/bin/bash
 2
 3 #                ref
 4 #                ---
 5
 6 # Rinomina (le) estensioni (dei) file.
 7 #
 8 #         ref vecchia_estensione nuova_estensione
 9 #
10 # Esempio:
11 # Per rinominare tutti i file *.gif della directory di lavoro in *.jpg,
12 #          ref gif jpg
13
14 ARG=2
15 E_ERR_ARG=65
16
17 if [ $# -ne "$ARG" ]
18 then
19   echo "Utilizzo: `basename $0` vecchia_estensione nuova_estensione"
20   exit $E_ERR_ARG
21 fi
22
23 for nomefile in *.$1
24 # Passa in rassegna l'elenco dei file che terminano con il 1mo argomento.
25 do
```

```
 26   mv $nomefile ${nomefile%$1}$2
 27   #  Toglie la parte di nomefile che verifica il 1mo argomento,
 28   #+ quindi aggiunge il 2do argomento.
 29 done
 30
 31 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 Esempio 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.

Esempio 9-18. Utilizzare la verifica di occorrenza per controllare stringhe arbitrarie

```
  1 #!/bin/bash
  2
  3 var1=abcd-1234-defg
  4 echo "var1 = $var1"
  5
  6 t=${var1#*-*}
  7 echo "var1 (viene tolto tutto ciò che si trova prima del primo"
  8 echo "trattino, compreso) = $t"
  9 #  t=${var1#*-}  Dà lo stesso risultato,
 10 #+ perché # verifica la stringa più corta,
 11 #+ e * verifica tutto quello che sta prima, compresa una stringa
vuota.
 12 # (Grazie, S. C. per la puntualizzazione.)
 13
 14 t=${var1##*-*}
 15 echo "Se var1 contiene un \"-\", viene restituita una stringa
vuota..."
 16 echo "var1 = $t"
```

```
 17
 18
 19 t=${var1%*-*}
 20 echo "var1 (viene tolto tutto ciò che si trova dopo l'ultimo"
 21 echo "trattino, compreso) = $t"
 22
 23 echo
 24
 25 # ----------------------------------------
 26 percorso=/home/bozo/idee/pensieri.di.oggi
 27 # ----------------------------------------
 28 echo "percorso = $percorso"
 29 t=${percorso##/*/}
 30 echo "percorso senza tutti i prefissi = $t"
 31 #  Stesso risultato con  t=`basename $percorso` , in questo caso
particolare.
 32 #  t=${percorso%/}; t=${t##*/}  è una soluzione più generica,
 33 #+ ma talvolta potrebbe non funzionare.
 34 #  Se $percorso termina con un carattere di ritorno a capo, allora
 35 #+ `basename $percorso` fallisce, al contrario dell'espressione
precedente.
 36 #  (Grazie, S.C.)
 37
 38 t=${percorso%/*.*}
 39 # Stesso risultato di   t=`dirname $percorso`
 40 echo "percorso a cui è stato tolto il suffisso (/pensieri.di.oggi)
= $t"
 41 #  Questi operatori possono non funzionare, come nei casi"../",
 42 #+ "/foo////", # "foo/", "/". Togliere i suffissi, specialmente
quando
 43 #+ basename non ne ha, ma dirname sì, complica la faccenda.
 44 # (Grazie, S.C.)
 45
 46 echo
 47
 48 t=${percorso:11}
 49 echo "$percorso, senza i primi 11 caratteri = $t"
 50 t=${percorso:11:5}
 51 echo "$percorso, senza i primi 11 caratteri e ridotto alla \
 52 lunghezza di 5 caratteri = $t"
 53
 54 echo
 55
 56 t=${percorso/bozo/clown}
 57 echo "$percorso con \"bozo\" sostituito da \"clown\" = $t"
 58 t=${percorso/oggi/}
 59 echo "$percorso con \"oggi\" cancellato = $t"
 60 t=${percorso//o/O}
 61 echo "$percorso con tutte le o minuscole cambiate in O maiuscole =
$t"
 62 t=${percorso//o/}
 63 echo "$percorso da cui sono state cancellate tutte le o = $t"
 64
 65 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*.

Esempio 9-19. Verifica di occorrenza di prefissi o suffissi di stringa

```
 1 #!/bin/bash
 2 # Sostituzione di occorrenza di prefisso/suffisso di stringa.
 3
 4 v0=abc1234zip1234abc    # Variabile originale.
 5 echo "v0 = $v0"         # abc1234zip1234abc
 6 echo
 7
 8 # Verifica del prefisso (inizio) della stringa.
 9 v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
10                         # |-|
11 echo "v1 = $v1"         # ABCDEF1234zip1234abc
12                         # |----|
13
14 # Verifica del suffisso (fine) della stringa.
15 v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
16                         #              |-|
17 echo "v2 = $v2"         # abc1234zip1234ABCDEF
18                         #               |----|
19
20 echo
21
22 #  -------------------------------------------------------
23 #  La verifica deve avvenire all'inizio/fine della stringa,
24 #+ altrimenti non verrà eseguita alcuna sostituzione.
25 #  -------------------------------------------------------
26 v3=${v0/#123/000}       # È verificata, ma non all'inizio.
27 echo "v3 = $v3"         # abc1234zip1234abc
28                         # NESSUNA SOSTITUZIONE.
29 v4=${v0/%123/000}       # È stata verificata, ma non alla fine.
30 echo "v4 = $v4"         # abc1234zip1234abc
31                         # NESSUNA SOSTITUZIONE.
32
33 exit 0
```

`${!prefissovar*}`, **`${!prefissovar@}`**

Verifica tutte le variabili precedentemente dichiarate i cui nomi iniziano con *prefissovar*.

```
 1 xyz23=qualsiasi_cosa
 2 xyz24=
 3
 4 a=${!xyz*}      #  Espande i nomi delle variabili dichiarate che
iniziano
 5                 #+ con "xyz".
 6 echo "a = $a"   # a = xyz23 xyz24
 7 a=${!xyz@}      # Come prima.
 8 echo "a = $a"   # a = xyz23 xyz24
 9
10 # La versione 2.04 di Bash possiede questa funzionalità.
```

Note

[1] Se $parametro è nullo, in uno script non interattivo, quest'ultimo viene terminato con exit status 127 (il codice di errore Bash di "command not found").

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)*

```
  1 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*

```
  1 declare -i numero
  2 # Lo script tratterà le successive occorrenze di "numero" come un
intero.
  3
  4 numero=3
  5 echo "numero = $numero"     # Numero = 3
  6
  7 numero=tre
  8 echo "Numero = $numero"     # numero = 0
  9 # 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.

```
  1 n=6/3
  2 echo "n = $n"       # n = 6/3
  3
  4 declare -i n
  5 n=6/3
  6 echo "n = $n"       # n = 2
```

-a *array*

```
  1 declare -a indici
```

La variabile `indici` verrà trattata come un array.

-f *funzioni*

```
  1 declare -f
```

In uno script, una riga con **`declare -f`** senza alcun argomento, elenca tutte le funzioni precedentemente definite in quello script.

```
  1 declare -f nome_funzione
```

Un **`declare -f nome_funzione`** elenca solo la funzione specificata.

-x export

```
1 declare -x var3
```

Dichiara la variabile come esportabile al di fuori dell'ambiente dello script stesso.

-x var=$valore

```
1 declare -x var3=373
```

Il comando **declare** consente di assegnare un valore alla variabile mentre viene dichiarata, impostando così anche le sue proprietà.

Esempio 9-20. Utilizzare declare per tipizzare le variabili

```
 1 #!/bin/bash
 2
 3 funz1 ()
 4 {
 5 echo Questa è una funzione.
 6 }
 7
 8 declare -f        # Elenca la funzione precedente.
 9
10 echo
11
12 declare -i var1   # var1 è un intero.
13 var1=2367
14 echo "var1 dichiarata come $var1"
15 var1=var1+1       #  La dichiarazione di intero elimina la necessità di
16                   #+ usare 'let'.
17 echo "var1 incrementata di 1 diventa $var1."
18 # Tentativo di modificare il valore di una variabile dichiarata come intero
19 echo "Tentativo di modificare var1 nel valore in virgola mobile 2367.1."
20 var1=2367.1       # Provoca un messaggio d'errore, la variabile non cambia.
21 echo "var1 è ancora $var1"
22
23 echo
24
25 declare -r var2=13.36         #  'declare' consente di impostare la
proprietà
26                               #+ della variabile e contemporaneamente
27                               #+ assegnarle un valore.
28 echo "var2 dichiarata come $var2"
29                               #  Tentativo di modificare una variabile in
sola
30                               #+ lettura.
31 var2=13.37                   #  Provoca un messaggio d'errore e l'uscita
dallo
32                               #+ script.
33
34 echo "var2 è ancora $var2"    # Questa riga non verrà eseguita.
35
36 exit 0                       # Lo script non esce in questo punto.
```

9.5. Referenziazione indiretta delle 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*.

Esempio 9-21. Referenziazioni indirette

```
 1 #!/bin/bash
 2 # Referenziazione indiretta a variabile.
 3
 4 a=lettera_alfabeto
 5 lettera_alfabeto=z
 6
 7 echo
 8
 9 # Referenziazione diretta.
10 echo "a = $a"
11
12 # Referenziazione indiretta.
13 eval a=\$$a
14 echo "Ora a = $a"
15
16 echo
17
18
19 # Proviamo a modificare la referenziazione di secondo ordine.
20
21 t=tabella_cella_3
22 tabella_cella_3=24
23 echo "\"tabella_cella_3\" = $tabella_cella_3"
24 echo -n "\"t\" dereferenziata = "; eval echo \$$t
25 # In questo caso, funziona anche
26 #   eval t=\$$t; echo "\"t\" = $t"
27 #  (perché?).
28
29 echo
30
31 t=tabella_cella_3
32 NUOVO_VAL=387
33 tabella_cella_3=$NUOVO_VAL
34 echo "Valore di \"tabella_cella_3\" modificato in $NUOVO_VAL."
35 echo "\"tabella_cella_3\" ora $tabella_cella_3"
36 echo -n "\"t\" dereferenziata  "; eval echo \$$t
37 # "eval" ha due argomenti "echo" e "\$$t" (impostata a $tabella_cella_3)
38 echo
39
40 # (Grazie, S.C., per aver chiarito il comportamento precedente.)
41
42
43 #  Un altro metodo è quello della notazione ${!t}, trattato nella
44 #+ sezione "Bash, versione 2". Vedi anche l'esempio "ex78.sh".
45
46 exit 0
```

Esempio 9-22. Passare una referenziazione indiretta a *awk*

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

```
 3 #  Altra versione dello script "column totaler"
 4 #+ che aggiunge una colonna (contenente numeri) nel file di destinazione.
 5 #  Viene utilizzata la referenziazione indiretta.
 6
 7 ARG=2
 8 E_ERR_ARG=65
 9
10 if [ $# -ne "$ARG" ] #  Verifica il corretto nr. di argomenti da riga
11                       #+ di comando.
12 then
13    echo "Utilizzo: `basename $0` nomefile numero_colonna"
14    exit $E_ERR_ARG
15 fi
16
17 nomefile=$1
18 numero_colonna=$2
19
20 #===== Fino a questo punto è uguale all'originale =====#
21
22
23 # Script awk di più di una riga vengono invocati con   awk ' ..... '
24
25
26 # Inizio script awk.
27 # -------------------------------------------------------
28 awk "
29
30 { totale += \$${numero_colonna} # referenziazione indiretta
31 }
32 END {
33      print totale
34      }
35
36      " "$nomefile"
37 # -----------------------------------------------
38 # Fine script awk.
39
40 #  La referenziazione indiretta evita le difficoltà della referenziazione
41 #+ di una variabile di shell all'interno di uno script awk incorporato.
42 #  Grazie, Stephane Chazelas.
43
44 exit 0
```

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 Esempio 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.

Esempio 9-23. Generare numeri casuali

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

```
 2
 3 # $RANDOM restituisce un intero casuale diverso ad ogni chiamata.
 4 # Intervallo nominale: 0 - 32767 (intero con segno di 16-bit).
 5
 6 NUM_MASSIMO=10
 7 contatore=1
 8
 9 echo
10 echo "$NUM_MASSIMO numeri casuali:"
11 echo "-----------------"
12 while [ "$contatore" -le $NUM_MASSIMO ]   #  Genera 10 ($NUM_MASSIMO)
13                                           #+ interi casuali.
14 do
15   numero=$RANDOM
16   echo $numero
17   let "contatore += 1"  # Incrementa il contatore.
18 done
19 echo "-----------------"
20
21 #  Se è necessario un intero casuale entro un dato intervallo, si usa
22 #+ l'operatore 'modulo', che restituisce il resto di una divisione.
23
24 INTERVALLO=500
25
26 echo
27
28 numero=$RANDOM
29 let "numero %= $INTERVALLO"
30 echo "Il numero casuale è inferiore a $INTERVALLO  ---  $numero"
31
32 echo
33
34 #  Se è necessario un intero casuale non inferiore a un certo limite,
35 #+ occorre impostare una verifica per eliminare tutti i numeri al di
36 #+ sotto di tale limite.
37
38 LIMITE_INFERIORE=200
39
40 numero=0   # inizializzazione
41 while [ "$numero" -le $LIMITE_INFERIORE ]
42 do
43   numero=$RANDOM
44 done
45 echo "Numero casuale maggiore di $LIMITE_INFERIORE ---  $numero"
46
47 echo
48
49 #  Le due tecniche precedenti possono essere combinate per ottenere un
50 #+ numero compreso tra due limiti.
51
52 numero=0   # inizializzazione
53 while [ "$numero" -le $LIMITE_INFERIORE ]
54 do
55   numero=$RANDOM
56   let "numero %= $INTERVALLO"  # Riduce $numero entro $INTERVALLO.
57 done
58 echo "Numero casuale tra $LIMITE_INFERIORE e $INTERVALLO ---  $numero"
59 echo
60
61
62 # Genera una scelta binaria, vale a dire, il valore "vero" o "falso".
63 BINARIO=2
```

```
numero=$RANDOM
T=1

let "numero %= $BINARIO"
#  Da notare che  let "numero >>= 14" dà una migliore distribuzione casuale
#+ (lo scorrimento a destra elimina tutto tranne l'ultima cifra binaria).
if [ "$numero" -eq $T ]
then
  echo "VERO"
else
  echo "FALSO"
fi

echo


# Si può simulare il lancio dei dadi.
MODULO=6  # Modulo 6 per un intervallo 0 - 5.
          # Aumentandolo di 1 si ottiene il desiderato intervallo 1 - 6.
          # Grazie a Paulo Marcel Coelho Aragao per la semplificazione.
dado1=0
dado2=0

#  Si lancia ciascun dado separatamente in modo da ottenere la corretta
#+ probabilità.

    let "dado1 = $RANDOM % $MODULO +1" # Lancio del primo dado.
    let "dado2 = $RANDOM % $MODULO +1" # Lancio del secondo dado.

let "punteggio = $dado1 + $dado2"
echo "Lancio dei dadi = $punteggio"
echo


exit 0
```

Esempio 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.


# Sceglie una carta, una qualsiasi.

Semi="Fiori
Quadri
Cuori
Picche"

Denominazioni="2
3
4
5
6
7
8
9
10
Fante
```

```
 24 Donna
 25 Re
 26 Asso"
 27
 28 seme=($Semi)                # Inizializza l'array.
 29 denominazione=($Denominazioni)
 30
 31 num_semi=${#seme[*]}        # Conta gli elementi dell'array.
 32 num_denominazioni=${#denominazione[*]}
 33
 34 echo -n "${denominazione[$((RANDOM%num_denominazioni))]} di "
 35 echo ${seme[$((RANDOM%num_semi))]}
 36
 37
 38 # $bozo sh pick-cards.sh
 39 # Fante di Fiori
 40
 41
 42 # Grazie, "jipe," per aver puntualizzato quest'uso di $RANDOM.
 43 exit 0
```

Jipe ha evidenziato una serie di tecniche per generare numeri casuali in un intervallo dato.

```
  1 #  Generare un numero casuale compreso tra 6 e 30.
  2 cnumero=$((RANDOM%25+6))
  3
  4 #  Generare un numero casuale, sempre nell'intervallo 6 - 30,
  5 #+ ma che deve essere divisibile per 3.
  6 cnumero=$(((RANDOM%30/3+1)*3))
  7
  8 #  È da notare che questo non sempre funziona.
  9 #  Fallisce quando $RANDOM restituisce 0.
 10
 11 #  Esercizio: Cercate di capire il funzionamento di questo esempio.
```

Bill Gradwohl ha elaborato una formula più perfezionata che funziona con i numeri positivi.

```
  1 cnumero=$(((RANDOM%(max-
min+divisibilePer))/divisibilePer*divisibilePer+min))
```

Qui Bill presenta una versatile funzione che restituisce un numero casuale compreso tra due valori specificati.

Esempio 9-25. Numero casuale in un intervallo dato

```
  1 #!/bin/bash
  2 # random-between.sh
  3 #  Numero casuale compreso tra due valori specificati.
  4 #  Script di Bill Gradwohl, con modifiche di secondaria importanza fatte
  5 #+ dall'autore del libro.
  6 #  Utilizzato con il permesso dell'autore.
  7
  8
  9 interCasuale() {
 10    #  Genera un numero casuale positivo o negativo
 11    #+ compreso tra $min e $max
 12    #+ e divisibile per $divisibilePer.
 13    #  Restituisce una distribuzione di valori "ragionevolmente casuale".
```

```
14    #
15    #  Bill Gradwohl - 1 Ott, 2003
16
17    sintassi() {
18    # Funzione inserita in un'altra.
19       echo
20       echo    "Sintassi: interCasuale [min] [max] [multiplo]"
21       echo
22       echo    "Si aspetta che vengano passati fino a 3 parametri,"
23       echo    "tutti però opzionali."
24       echo    "min è il valore minimo"
25       echo    "max è il valore massimo"
26       echo    "multiplo specifica che il numero generato deve essere un"
27       echo    "multiplo di questo valore."
28       echo    "    cioè divisibile esattamente per questo numero."
29       echo
30       echo    "Se si omette qualche valore, vengono usati"
31       echo    "quelli preimpostati: 0 32767 1"
32       echo    "L'esecuzione senza errori restituisce 0, altrimenti viene"
33       echo    "richiamata la funzione sintassi e restituito 1."
34       echo    "Il numero generato viene restituito nella variabile globale"
35       echo    "interCasualeNum"
36       echo    "Valori negativi passati come parametri vengono gestiti"
37       echo    "correttamente."
38    }
39
40    local min=${1:-0}
41    local max=${2:-32767}
42    local divisibilePer=${3:-1}
43    #  Assegnazione dei valori preimpostati, nel caso di mancato passaggio
44    #+ dei parametri alla funzione.
45
46    local x
47    local intervallo
48
49    # Verifica che il valore di divisibilePer sia positivo.
50    [ ${divisibilePer} -lt 0 ] && divisibilePer=$((0-divisibilePer))
51
52    # Controllo di sicurezza.
53    if [ $# -gt 3 -o ${divisibilePer} -eq 0 -o  ${min} -eq ${max} ]; then
54       sintassi
55       return 1
56    fi
57
58    # Verifica se min e max sono scambiati.
59    if [ ${min} -gt ${max} ]; then
60       # Li scambia.
61       x=${min}
62       min=${max}
63       max=${x}
64    fi
65
66    #  Se min non è esattamente divisibile per $divisibilePer,
67    #+ viene ricalcolato.
68    if [ $((min/divisibilePer*divisibilePer)) -ne ${min} ]; then
69       if [ ${min} -lt 0 ]; then
70          min=$((min/divisibilePer*divisibilePer))
71       else
72          min=$((((min/divisibilePer)+1)*divisibilePer))
73       fi
74    fi
75
```

```
 76    #  Se max non è esattamente divisibile per $divisibilePer,
 77    #+ viene ricalcolato.
 78    if [ $((max/divisibilePer*divisibilePer)) -ne ${max} ]; then
 79       if [ ${max} -lt 0 ]; then
 80          max=$((((max/divisibilePer)-1)*divisibilePer))
 81       else
 82          max=$((max/divisibilePer*divisibilePer))
 83       fi
 84    fi
 85
 86    #  ----------------------------------------------------------------------
-
 87    #  Ora il lavoro vero.
 88
 89    #  E' da notare che per ottenere una corretta distribuzione dei valori
 90    #+ estremi, si deve agire su un intervallo che va da 0 a
 91    #+ abs(max-min)+divisibilePer, non semplicemente abs(max-min)+1.
 92
 93    #  Il leggero incremento produrrà la giusta distribuzione per i
 94    #+ valori limite.
 95
 96    #  Se si cambia la formula e si usa abs(max-min)+1 si otterranno ancora
 97    #+ dei risultati corretti, ma la loro casualità sarà falsata
 98    #+ dal fatto che il numero di volte in cui verranno restituiti gli
estremi
 99    #+ ($min e $max) sarà considerevolmente inferiore a quella ottenuta
100    #+ usando la formula corretta.
101    #  ----------------------------------------------------------------------
-
102
103    intervallo=$((max-min))
104    [ ${intervallo} -lt 0 ] && intervallo=$((0-intervallo))
105    let intervallo+=divisibilePer
106    interCasualeNum=$(((RANDOM%intervallo)/divisibilePer*divisibilePer+min))
107
108    return 0
109
110    #  Tuttavia, Paulo Marcel Coelho Aragao sottolinea che
111    #+ quando $max e $min non sono divisibili per $divisibilePer,
112    #+ la formula sbaglia.
113    #
114    #  Suggerisce invece la seguente:
115    #    numeroc = $(((RANDOM%(max-min+1)+min)/divisibilePer*divisibilePer))
116 }
117
118 # Verifichiamo la funzione.
119
120 min=-14
121 max=20
122 divisibilePer=3
123
124
125 #  Genera un array e controlla che si sia ottenuto almeno uno dei risultati
126 #+ possibili, se si effettua un numero sufficiente di tentativi.
127
128 declare -a risultati
129 minimo=${min}
130 massimo=${max}
131    if [ $((minimo/divisibilePer*divisibilePer)) -ne ${minimo} ]; then
132       if [ ${minimo} -lt 0 ]; then
133          minimo=$((minimo/divisibilePer*divisibilePer))
134       else
```

```
135          minimo=$((((minimo/divisibilePer)+1)*divisibilePer))
136       fi
137    fi
138
139
140    #  Se max non è esattamente divisibile per $divisibilePer,
141    #+ viene ricalcolato.
142
143    if [ $((massimo/divisibilePer*divisibilePer)) -ne ${massimo} ]; then
144       if [ ${massimo} -lt 0 ]; then
145          massimo=$((((massimo/divisibilePer)-1)*divisibilePer))
146       else
147          massimo=$((massimo/divisibilePer*divisibilePer))
148       fi
149    fi
150
151
152 #  Poiché gli indici degli array possono avere solo valori positivi,
153 #+ è necessario uno spiazzamento che garantisca il raggiungimento
154 #+ di questo risultato.
155
156 spiazzamento=$((0-minimo))
157 for ((i=${minimo}; i<=${massimo}; i+=divisibilePer)); do
158    risultati[i+spiazzamento]=0
159 done
160
161
162 # Ora si esegue per un elevato numero di volte, per vedere cosa si ottiene.
163 nr_volte=1000  #  L'autore dello script suggeriva 100000,
164               #+ ma sarebbe occorso veramente molto tempo.
165
166 for ((i=0; i<${nr_volte}; ++i)); do
167
168    #  Notate che qui min e max sono specificate in ordine inverso
169    #+ per vedere, in questo caso, il corretto comportamento della funzione.
170
171    interCasuale ${max} ${min} ${divisibilePer}
172
173    # Riporta un errore se si verifica un risultato inatteso.
174    [ ${interCasualeNum} -lt ${min} -o ${interCasualeNum} -gt ${max} ] \
175 && echo errore MIN o MAX  - ${interCasualeNum}!
176    [ $((interCasualeNum%${divisibilePer})) -ne 0 ] \
177 && echo DIVISIBILE PER errore - ${interCasualeNum}!
178
179    # Registra i risultati statisticamente.
180    risultati[interCasualeNum+spiazzamento]=\
181 $((risultati[interCasualeNum+spiazzamento]+1))
182 done
183
184
185
186 # Controllo dei risultati
187
188 for ((i=${minimo}; i<=${massimo}; i+=divisibilePer)); do
189    [ ${risultati[i+spiazzamento]} -eq 0 ] && echo "Nessun risultato per $i."
\
190 || echo "${i} generato ${risultati[i+spiazzamento]} volte."
191 done
192
193
194 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...

Esempio 9-26. Lanciare un dado con RANDOM

```
  1 #!/bin/bash
  2 # Quant'è casuale RANDOM?
  3
  4 RANDOM=$$        #  Cambia il seme del generatore di numeri
  5                  #+ casuali usando l'ID di processo dello script.
  6
  7 FACCE=6          # Un dado ha 6 facce.
  8 NUMMAX_LANCI=600 # Aumentatelo, se non avete nient'altro di meglio da fare.
  9 lanci=0          # Contatore dei lanci.
 10
 11 tot_uno=0        #  I contatori devono essere inizializzati a 0 perché
 12 tot_due=0        #+ una variabile non inizializzata ha valore nullo, non
zero.
 13 tot_tre=0
 14 tot_quattro=0
 15 tot_cinque=0
 16 tot_sei=0
 17
 18 visualizza_risultati ()
 19 {
 20 echo
 21 echo "totale degli uno = $tot_uno"
 22 echo "totale dei due = $tot_due"
 23 echo "totale dei tre = $tot_tre"
 24 echo "totale dei quattro = $tot_quattro"
 25 echo "totale dei cinque = $tot_cinque"
 26 echo "totale dei sei = $tot_sei"
 27 echo
 28 }
 29
 30 aggiorna_contatori()
 31 {
 32 case "$1" in
 33   0) let "tot_uno += 1";;   #  Poiché un dado non ha lo "zero",
 34                             #+ lo facciamo corrispondere a 1.
 35   1) let "tot_due += 1";;   #  1 a 2, ecc.
 36   2) let "tot_tre += 1";;
 37   3) let "tot_quattro += 1";;
 38   4) let "tot_cinque += 1";;
 39   5) let "tot_sei += 1";;
 40 esac
 41 }
 42
 43 echo
 44
 45
 46 while [ "$lanci" -lt "$NUMMAX_LANCI" ]
 47 do
 48   let "dado1 = RANDOM % $FACCE"
 49   aggiorna_contatori $dado1
 50   let "lanci += 1"
 51 done
 52
 53 visualizza_risultati
 54
```

```
 55 #  I punteggi dovrebbero essere distribuiti abbastanza equamente,
nell'ipotesi
 56 #+ che RANDOM sia veramente casuale.
 57 #  Con $NUMMAX_LANCI impostata a 600, la frequenza di ognuno dei sei numeri
 58 #+ dovrebbe aggirarsi attorno a 100, più o meno 20 circa.
 59 #
 60 #  Ricordate che RANDOM è un generatore pseudocasuale, e neanche
 61 #+ particolarmente valido.
 62
 63 #  La casualità è un argomento esteso e complesso.
 64 #  Sequenze "casuali" sufficientemente lunghe possono mostrare
 65 #+ un andamento caotico e "non-casuale".
 66
 67 # Esercizio (facile):
 68 # ------------------
 69 # Riscrivete lo script per simulare il lancio di una moneta 1000 volte.
 70 # Le possibilità sono "TESTA" o "CROCE".
 71
 72 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.)

Esempio 9-27. Cambiare il seme di RANDOM

```
  1 #!/bin/bash
  2 # seeding-random.sh: Cambiare il seme della variabile RANDOM.
  3
  4 MAX_NUMERI=25     # Quantità di numeri che devono essere generati.
  5
  6 numeri_casuali ()
  7 {
  8 contatore=0
  9 while [ "$contatore" -lt "$MAX_NUMERI" ]
 10 do
 11   numero=$RANDOM
 12   echo -n "$numero "
 13   let "contatore += 1"
 14 done
 15 }
 16
 17 echo; echo
 18
 19 RANDOM=1          # Impostazione del seme di RANDOM.
 20 numeri_casuali
 21
 22 echo; echo
 23
 24 RANDOM=1          # Stesso seme...
 25 numeri_casuali    # ...riproduce esattamente la serie precedente.
 26                   #
 27                   # Ma, quant'è utile duplicare una serie di numeri
"casuali"?
 28
 29 echo; echo
 30
 31 RANDOM=2          # Altro tentativo, ma con seme diverso...
 32 numeri_casuali    # viene generata una serie differente.
 33
```

```
34 echo; echo
35
36 # RANDOM=$$  imposta il seme di RANDOM all'id di processo dello script.
37 # È anche possibile usare come seme di RANDOM i comandi 'time' o 'date'.
38
39 # Ancora più elegante...
40 SEME=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
41 #  Output pseudocasuale prelevato da /dev/urandom (file di
42 #+ dispositivo di sistema pseudo-casuale), quindi convertito
43 #+ con "od" in una riga di numeri (ottali) visualizzabili,
44 #+ infine "awk" ne recupera solamente uno per SEME.
45
46 RANDOM=$SEME
47 numeri_casuali
48
49 echo; echo
50
51 exit 0
```

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 Esempio 12-50).

Ci sono altri metodi per generare numeri pseudocasuali in uno script. **Awk** ne fornisce uno molto comodo.

Esempio 9-28. Numeri pseudocasuali utilizzando awk

```
  1 #!/bin/bash
  2 # random2.sh: Restituisce un numero pseudo-casuale nell'intervallo 0 -
1.
  3 # Uso della funzione awk rand().
  4
  5 SCRIPTAWK=' { srand(); print rand() } '
  6 #            Comando(i) / parametri passati ad awk
  7 # Notate che srand() ricalcola il seme del generatore di numeri di awk.
  8
  9
 10 echo -n "Numeri casuali tra 0 e 1 = "
 11
 12 echo | awk "$SCRIPTAWK"
 13 # Cosa succede se si omette 'echo'?
 14
 15 exit 0
 16
 17
 18 # Esercizi:
 19 # ---------
 20
 21 # 1) Usando un ciclo, visualizzare 10 differenti numeri casuali.
 22 #    (Suggerimento: bisogna ricalcolare un diverso seme per la funzione
 23 #+   "srand()" ad ogni passo del ciclo. Cosa succede se non viene
fatto?)
 24
 25 # 2) Usando come fattore di scala un multiplo intero, generare numeri
 26 #+   casuali nell'intervallo tra 10 e 100.
 27
 28 # 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.

Esempio 9-29. Gestire le variabili in stile C

```
 1 #!/bin/bash
 2 # Manipolare una variabile, in stile C, usando il costrutto ((...)).
 3
 4
 5 echo
 6
 7 (( a = 23 ))  #  Impostazione, in stile C, con gli spazi da entrambi i lati
 8               #+ dell' "=".
 9 echo "a (valore iniziale) = $a"
10
11 (( a++ ))     # Post-incremento di 'a', stile C.
12 echo "a (dopo a++) = $a"
13
14 (( a-- ))     # Post-decremento di 'a', stile C.
15 echo "a (dopo a--) = $a"
16
17 (( ++a ))     # Pre-incremento di 'a', stile C.
18 echo "a (dopo ++a) = $a"
19
20 (( --a ))     # Pre-decremento di 'a', stile C.
21 echo "a (dopo --a) = $a"
22
23 echo
24
25 (( t = a<45?7:11 ))   # Operatore ternario del C.
26 echo "Se a < 45, allora t = 7, altrimenti t = 11."
27 echo "t = $t "        # Si!
28
29 echo
30
31
32 # ---------------------
33 # Attenzione, sorpresa!
34 # ---------------------
35 #  Evidentemente Chet Ramey ha contrabbandato un mucchio di costrutti in
36 #+ stile C, non documentati, in Bash (in realtà adattati da ksh, in
37 #+ quantità notevole).
38 #  Nella documentazione Bash, Ramey chiama ((...)) matematica di shell,
39 #+ ma ciò va ben oltre l'aritmetica.
40 #  Mi spiace, Chet, ora il segreto è svelato.
41
42 # Vedi anche l'uso del costrutto ((...)) nei cicli "for" e "while".
43
44 # Questo costrutto funziona solo nella versione  2.04 e successive, di Bash.
45
46 exit 0
```

Vedi anche Esempio 10-12.

Capitolo 10. Cicli ed alternative

Sommario

Le operazioni sui blocchi di codice sono la chiave per creare 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

Ad ogni passo del ciclo, *arg* assume il valore di ognuna delle successive variabili elencate in *lista*.

```
1 for arg in "$var1" "$var2" "$var3" ... "$varN"
2 # Al 1° passo del ciclo, $arg = $var1
3 # Al 2° passo del ciclo, $arg = $var2
4 # Al 3° passo del ciclo, $arg = $var3
5 # ...
6 # Al passo N° del ciclo, $arg = $varN
7
8 #  Bisogna applicare il "quoting" agli argomenti della [lista] per
9 #+ 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

Esempio 10-1. Semplici cicli for

```
  1 #!/bin/bash
  2 # Elenco di pianeti.
  3
  4 for pianeta in Mercurio Venere Terra Marte Giove Saturno Urano
Nettuno Plutone
  5 do
  6   echo $pianeta  # Ogni pianeta su una riga diversa
  7 done
  8
  9 echo
 10
 11 for pianeta in "Mercurio Venere Terra Marte Giove Saturno Urano
Nettuno Plutone"
 12 # Tutti i pianeti su un'unica riga.
 13 # L'intera "lista" racchiusa tra apici doppi crea un'unica
variabile.
 14 do
 15   echo $pianeta
 16 done
 17
 18 exit 0
```

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 Esempio 11-14) per forzare la verifica di ciascun elemento in **[lista]** e per assegnare ad ogni componente i rispettivi parametri posizionali.

Esempio 10-2. Ciclo for con due parametri in ogni elemento [lista]

```
  1 #!/bin/bash
  2 # Pianeti rivisitati.
  3
  4 # Associa il nome di ogni pianeta con la sua distanza dal sole.
  5
  6 for pianeta in "Mercurio 36" "Venere 67" "Terra 93"  "Marte 142"
"Giove 483"
  7 do
  8   set -- $pianeta  #  Verifica la variabile "pianeta" e imposta i
parametri
  9                    #+ posizionali.
 10   #  i "--" evitano sgradevoli sorprese nel caso $pianeta sia nulla
 11   #+ o inizi con un trattino.
 12
 13   #  Potrebbe essere necessario salvare i parametri posizionali
 14   #+ originari, perché vengono sovrascritti.
 15   #  Un modo per farlo è usare un array,
 16   #        param_origin=("$@")
 17
 18   echo "$1         $2,000,000 miglia dal sole"
 19
 20   ##-------due  tab---- servono a concatenare gli zeri al parametro
$2
 21 done
 22
 23 # (Grazie, S.C., per i chiarimenti aggiuntivi.)
 24
 25 exit 0
```

In un ciclo **for**, una variabile può sostituire **[lista]**.

Esempio 10-3. *Fileinfo:* operare su un elenco di file contenuto in una variabile

```
 1 #!/bin/bash
 2 # fileinfo.sh
 3
 4 FILE="/usr/sbin/privatepw
 5 /usr/sbin/pwck
 6 /usr/sbin/go500gw
 7 /usr/bin/fakefile
 8 /sbin/mkreiserfs
 9 /sbin/ypbind"     # Elenco dei file sui quali volete informazioni.
10                   # Compreso il falso file /usr/bin/fakefile.
11
12 echo
13
14 for file in $FILE
15 do
16
17   if [ ! -e "$file" ]       # Verifica se il file esiste.
18   then
19     echo "$file non esiste."; echo
20     continue                # Verifica il successivo.
21    fi
22
23   ls -l $file | awk '{ print $9 "         dimensione file: " $5 }'
24   # Visualizza 2 campi.
25
26   whatis `basename $file`   # Informazioni sul file.
27   echo
28 done
29
30 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.

Esempio 10-4. Agire sui file con un ciclo for

```
 1 #!/bin/bash
 2 # list-glob.sh: Generare [lista] in un ciclo for usando il
"globbing".
 3
 4 echo
 5
 6 for file in *
 7 do
 8   ls -l "$file"  # Elenca tutti i file in $PWD (directory corrente).
 9   #  Ricordate che il carattere jolly  "*" verifica tutti i file,
10   #+ tuttavia, il "globbing" non verifica i file i cui nomi iniziano
11   #+ con un punto.
12
13   #  Se il modello non verifica nessun file, allora si autoespande.
14   #  Per evitarlo impostate l'opzione nullglob (shopt -s nullglob).
15   # Grazie, S.C.
16 done
17
18 echo; echo
19
20 for file in [jx]*
21 do
```

```
 22   rm -f $file    #  Cancella solo i file i cui nomi iniziano con
 23                  #+ "j" o "x" presenti in $PWD.
 24   echo "Rimosso il file \"$file\"".
 25 done
 26
 27 echo
 28
 29 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'Esempio A-17.

Esempio 10-5. Tralasciare `in [lista]` in un ciclo for

```
  1 #!/bin/bash
  2
  3 # Invocate lo script sia con che senza argomenti e osservate cosa
succede.
  4
  5 for a
  6 do
  7  echo -n "$a "
  8 done
  9
 10 #  Manca 'in lista', quindi il ciclo opera su '$@'
 11 #+ (elenco degli argomenti da riga di comando, compresi gli spazi).
 12
 13 echo
 14
 15 exit 0
```

È possibile impiegare la sostituzione di comando per generare **`[lista]`**. Vedi anche Esempio 12-45, Esempio 10-10 ed Esempio 12-39.

Esempio 10-6. Generare [lista] in un ciclo for con la sostituzione di comando

```
  1 #!/bin/bash
  2 #  for-loopcmd.sh: un ciclo for con [lista]
  3 #+ prodotta dalla sostituzione di comando.
  4
  5 NUMERI="9 7 3 8 37.53"
  6
  7 for numero in `echo $NUMERI`  # for numero in 9 7 3 8 37.53
  8 do
  9   echo -n "$numero "
 10 done
 11
 12 echo
 13 exit 0
```

Ecco un esempio un po' più complesso dell'utilizzo della sostituzione di comando per creare [lista].

Esempio 10-7. Un'alternativa con grep per i file binari

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

```
  2 # bin-grep.sh:  Localizza le stringhe in un file binario.
  3
  4 # Un'alternativa con "grep" per file binari.
  5 # Effetto simile a "grep -a"
  6
  7 E_ERR_ARG=65
  8 E_NOFILE=66
  9
 10 if [ $# -ne 2 ]
 11 then
 12   echo "Utilizzo: `basename $0` stringa_di_ricerca nomefile"
 13   exit $E_ERR_ARG
 14 fi
 15
 16 if [ ! -f "$2" ]
 17 then
 18   echo "Il file \"$2\" non esiste."
 19   exit $E_NOFILE
 20 fi
 21
 22
 23 IFS="\n"         # Su suggerimento di Paulo Marcel Coelho Aragao.
 24 for parola in $( strings "$2" | grep "$1" )
 25 # Il comando "strings" elenca le stringhe nei file binari.
 26 # L'output viene collegato (pipe) a "grep" che verifica la stringa
cercata.
 27 do
 28   echo $parola
 29 done
 30
 31 #  Come ha sottolineato S.C., le righe 23 - 29 potrebbero essere
 32 #+ sostituite con la più semplice
 33 #    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'
 34
 35
 36 #  Provate qualcosa come  "./bin-grep.sh mem /bin/ls"  per
esercitarvi
 37 #+ con questo script.
 38
 39 exit 0
```

Sempre sullo stesso tema.

Esempio 10-8. Elencare tutti gli utenti del sistema

```
  1 #!/bin/bash
  2 # userlist.sh
  3
  4 FILE_PASSWORD=/etc/passwd
  5 n=1           # Numero utente
  6
  7 for nome in $(awk 'BEGIN{FS=":"}{print $1}' < "$FILE_PASSWORD" )
  8 # Separatore di campo = :^^^^^^
  9 # Visualizza il primo campo      ^^^^^^^^
 10 # Ottiene l'input dal file delle password    ^^^^^^^^^^^^^^^^^
 11 do
 12   echo "UTENTE nr.$n = $nome"
 13   let "n += 1"
 14 done
 15
 16
```

```
 17 # UTENTE nr.1 = root
 18 # UTENTE nr.2 = bin
 19 # UTENTE nr.3 = daemon
 20 # ...
 21 # UTENTE nr.30 = bozo
 22
 23 exit 0
```

Esempio finale di [lista] risultante dalla sostituzione di comando.

Esempio 10-9. Verificare tutti i file binari di una directory in cerca degli autori

```
  1 #!/bin/bash
  2 # findstring.sh:
  3 # Cerca una stringa particolare nei binari di una directory
specificata.
  4
  5 directory=/usr/bin/
  6 stringa="Free Software Foundation"  # Vede quali file sono della
FSF.
  7
  8 for file in $( find $directory -type f -name '*' | sort )
  9 do
 10   strings -f $file | grep "$stringa" | sed -e "s%$directory%%"
 11   #  Nell'espressione  "sed", è necessario sostituire il normale
 12   #+ delimitatore "/" perché si dà il caso che "/" sia uno dei
 13   #+ caratteri che deve essere filtrato.
 14 done
 15
 16 exit 0
 17
 18 #  Esercizio (facile):
 19 #  ------------------
 20 #  Modificate lo script in modo tale che accetti come parametri da
 21 #+ 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.

Esempio 10-10. Elencare i link simbolici presenti in una directory

```
  1 #!/bin/bash
  2 # symlinks.sh: Elenca i link simbolici presenti in una directory.
  3
  4
  5 directory=${1-`pwd`}
  6 #  Imposta come predefinita la directory di lavoro corrente, nel
caso non ne
  7 #+ venga specificata alcuna.
  8 #  Corrisponde al seguente blocco di codice.
  9 # ----------------------------------------------------------------
-
 10 # ARG=1                # Si aspetta un argomento da riga di
comando.
 11 #
 12 # if [ $# -ne "$ARG" ]  # Se non c'è 1 argomento...
 13 # then
 14 #   directory=`pwd`     # directory di lavoro corrente
 15 # else
```

```
 16 #   directory=$1
 17 # fi
 18 # ------------------------------------------------------------------
-
 19
 20 echo "Link simbolici nella directory \"$directory\""
 21
 22 for file in "$( find $directory -type l )"  # -type l = link
simbolici
 23 do
 24   echo "$file"
 25 done | sort                                 #  Se manca sort,
l'elenco
 26                                            #+ non verrà ordinato.
 27 #  Per essere precisi, in realtà in questo caso un ciclo non sarebbe
necessario,
 28 #+ perchè l'output del comando "find" viene espanso in un'unica
parola.
 29 #  Tuttavia, illustra bene questa modalità e ne facilita la
comprensione.
 30
 31 #  Come ha evidenziato Dominik 'Aeneas' Schnitzer,
 32 #+ se non si usa il "quoting" per $( find $directory -type l ) i
nomi dei
 33 #+ file contenenti spazi non vengono visualizzati correttamente.
 34 #  Il nome viene troncato al primo spazio incontrato.
 35
 36 exit 0
 37
 38
 39 # Jean Helou propone la seguente alternativa:
 40
 41 echo "Link simbolici nella directory \"$directory\""
 42 # Salva l'IFS corrente. Non si è mai troppo prudenti.
 43 VECCHIOIFS=$IFS
 44 IFS=:
 45
 46 for file in $(find $directory -type l -printf "%p$IFS")
 47 do     #                              ^^^^^^^^^^^^^^^^
 48        echo "$file"
 49        done|sort
```

Lo `stdout` di un ciclo può essere rediretto in un file, come dimostra la piccola modifica apportata all'esempio precedente.

Esempio 10-11. Link simbolici presenti in una directory salvati in un file

```
  1 #!/bin/bash
  2 # symlinks.sh: Elenca i link simbolici presenti in una directory.
  3
  4 OUTFILE=symlinks.list                        # file di
memorizzazione
  5
  6 directory=${1-`pwd`}
  7 #  Imposta come predefinita la directory di lavoro corrente, nel
caso non
  8 #+ ne venga specificata alcuna.
  9
 10
 11 echo "Link simbolici nella directory \"$directory\"" > "$OUTFILE"
```

```
 12 echo "--------------------------" >> "$OUTFILE"
 13
 14 for file in "$( find $directory -type l )"    # -type l = link
simbolici
 15 do
 16   echo "$file"
 17 done | sort >> "$OUTFILE"                     #  stdout del ciclo
rediretto
 18 #           ^^^^^^^^^^^^                        al file di
memorizzazione.
 19
 20 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.

Esempio 10-12. Un ciclo for in stile C

```
  1 #!/bin/bash
  2 # Due modi per contare fino a 10.
  3
  4 echo
  5
  6 # Sintassi standard.
  7 for a in 1 2 3 4 5 6 7 8 9 10
  8 do
  9   echo -n "$a "
 10 done
 11
 12 echo; echo
 13
 14 #

15 +=====================================================================+
 16
 17 # Ora facciamo la stessa cosa usando la sintassi in stile C.
 18
 19 LIMITE=10
 20
 21 for ((a=1; a <= LIMITE; a++))  # Doppie parentesi, e "LIMITE" senza
"$".
 22 do
 23   echo -n "$a "
 24 done                           # Un costrutto preso in prestito da
'ksh93'.
 25
 26 echo; echo
 27
 28 #
+=====================================================================+
 29
 30 #  Uso dell' "operatore virgola" del C per incrementare due
variabili
 31 #+ contemporaneamente.
 32
 33 for ((a=1, b=1; a <= LIMITE; a++, b++))  # La virgola concatena le
operazioni.
 34 do
 35   echo -n "$a-$b "
 36 done
```

```
 37
 38 echo; echo
 39
 40 exit 0
```

Vedi anche Esempio 26-15, Esempio 26-16 e Esempio A-7.

Adesso un *ciclo for* impiegato in un'applicazione "pratica".

Esempio 10-13. Utilizzare efax in modalità batch

```
  1 #!/bin/bash
  2
  3 ARG_ATTESI=2
  4 E_ERR_ARG=65
  5
  6 if [ $# -ne $ARG_ATTESI ]
  7 # Verifica il corretto numero di argomenti.
  8 then
  9    echo "Utilizzo: `basename $0` nr_telefono file_testo"
 10    exit $E_ERR_ARG
 11 fi
 12
 13
 14 if [ ! -f "$2" ]
 15 then
 16   echo "Il file $2 non è un file di testo"
 17   exit $E_ERR_ARG
 18 fi
 19
 20
 21 fax make $2              # Crea file fax formattati dai file di
testo.
 22
 23 for file in $(ls $2.0*)  # Concatena i file appena creati.
 24                          # Usa il carattere jolly in lista.
 25 do
 26   fil="$fil $file"
 27 done
 28
 29 efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # Esegue il lavoro.
 30
 31
 32 # Come ha sottolineato S.C. il ciclo for potrebbe essere sostituito
con
 33 #    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
 34 # ma non sarebbe stato altrettanto istruttivo [sorriso].
 35
 36 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.

Esempio 10-14. Un semplice ciclo while

```
  1 #!/bin/bash
  2
  3 var0=0
  4 LIMITE=10
  5
  6 while [ "$var0" -lt "$LIMITE" ]
  7 do
  8   echo -n "$var0 "        # -n sopprime il ritorno a capo.
  9   #             ^           Lo spazio serve a separare i numeri
visualizzati.
 10   var0=`expr $var0 + 1`   # var0=$(($var0+1))  anche questa forma va
bene.
 11                           # var0=$((var0 + 1)) anche questa forma va
bene.
 12                           # let "var0 += 1"    anche questa forma va
bene.
 13 done                      # Anche vari altri metodi funzionano.
 14
 15 echo
 16
 17 exit 0
```

Esempio 10-15. Un altro ciclo while

```
  1 #!/bin/bash
  2
  3 echo
  4
  5 while [ "$var1" != "fine" ]     #  while test "$var1" != "fine"
  6 do                              #  altra forma valida.
  7   echo "Immetti la variabile #1 (fine per terminare) "
  8   read var1                     #  Non 'read $var1' (perché?).
  9   echo "variabile #1 = $var1"   #  È necessario il "quoting"
 10                                 #+ per la presenza di "#".
 11   # Se l'input è 'fine', viene visualizzato a questo punto.
 12   # La verifica per l'interruzione del ciclo, infatti, è posta
all'inizio.
 13
 14   echo
 15 done
 16
 17 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.

Esempio 10-16. Ciclo while con condizioni multiple

```
  1 #!/bin/bash
  2
  3 var1=nonimpostata
  4 precedente=$var1
  5
  6 while echo "variabile-precedente = $precedente"
  7       echo
  8       precedente=$var1
  9       [ "$var1" != fine ] # Tiene traccia del precedente valore di
$var1.
 10       #  "while" con quattro condizioni, ma è solo l'ultima che
controlla
 11       #+ il ciclo.
 12       #  È l'*ultimo* exit status quello che conta.
 13 do
 14 echo "Immetti la variable nr.1 (fine per terminare) "
 15   read var1
 16   echo "variabile nr.1 = $var1"
 17 done
 18
 19 # Cercate di capire come tutto questo funziona.
 20 # È un tantino complicato.
 21
 22 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 Esempio 9-29).

Esempio 10-17. Sintassi in stile C di un ciclo while

```
  1 #!/bin/bash
  2 # wh-loopc.sh: Contare fino a 10 con un ciclo "while".
  3
  4 LIMITE=10
  5 a=1
  6
  7 while [ "$a" -le $LIMITE ]
  8 do
  9   echo -n "$a "
 10   let "a+=1"
 11 done           # Fin qui nessuna novità.
 12
 13 echo; echo
 14
 15 #
+==========================================================================+
 16
 17 # Rifatto con la sintassi del C.
 18
 19 ((a = 1))      # a=1
 20 #  Le doppie parentesi consentono gli spazi nell'impostazione di una
 21 #+ variabile, come in C.
 22
```

```
23 while (( a <= LIMITE ))   #  Doppie parentesi senza "$" che precede
24                           #+ il nome della variabile.
25 do
26   echo -n "$a "
27   ((a += 1))   # let "a+=1"
28   #  Si.
29   #  Le doppie parentesi consentono di incrementare una variabile
30   #+ con la sintassi del C.
31 done
32
33 echo
34
35 # Ora i programmatori in C si sentiranno a casa loro anche con Bash.
36
37 exit 0
```

Un ciclo **while** può avere il proprio `stdin` redirettо da un file tramite un < alla fine del blocco.

In un ciclo **while** il relativo `stdin` può essere fornito da una pipe.

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

Esempio 10-18. Ciclo until

```
 1 #!/bin/bash
 2
 3 CONDIZIONE_CONCLUSIONE=fine
 4
 5 until [ "$var1" = "$CONDIZIONE_CONCLUSIONE" ]
 6 # Condizione di verifica all'inizio del ciclo.
 7 do
 8   echo "Immetti variabile nr.1 "
 9   echo "($CONDIZIONE_CONCLUSIONE per terminare)"
10   read var1
11   echo "variabile nr.1 = $var1"
12   echo
13 done
14
15 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.

Esempio 10-19. Cicli annidati

```
 1 #!/bin/bash
 2 # Cicli "for" annidati.
 3
 4 esterno=1             # Imposta il contatore del ciclo esterno.
 5
 6 # Inizio del ciclo esterno.
 7 for a in 1 2 3 4 5
 8 do
 9   echo "Passo $esterno del ciclo esterno."
10   echo "---------------------"
11   interno=1           # Imposta il contatore del ciclo interno.
12
13   # Inizio del ciclo interno.
14   for b in 1 2 3 4 5
15   do
16     echo "Passo $interno del ciclo interno."
17     let "interno+=1"  # Incrementa il contatore del ciclo interno.
18   done
19   # Fine del ciclo interno.
20
21   let "esterno+=1"    #  Incrementa il contatore del ciclo esterno.
22   echo                #  Spaziatura tra gli output dei successivi
23                       #+ passi del ciclo esterno.
24 done
25 # Fine del ciclo esterno.
26
27 exit 0
```

Vedi Esempio 26-11 per un'illustrazione di cicli "while" annidati e Esempio 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.

Esempio 10-20. Effetti di break e continue in un ciclo

```
 1 #!/bin/bash
 2
 3 LIMITE=19  # Limite superiore
 4
 5 echo
 6 echo "Visualizza i numeri da 1 fino a 20 (saltando 3 e 11)."
 7
 8 a=0
 9
10 while [ $a -le "$LIMITE" ]
11 do
12  a=$(($a+1))
13
14  if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  #  Esclude  3 e 11
15  then
16    continue  #  Salta la parte restante di questa particolare
17              #+ iterazione del ciclo.
18  fi
19
20  echo -n "$a "
21 done
22
23 # Esercizio:
24 # Perché il ciclo visualizza fino a 20?
25
26 echo; echo
27
28 echo Visualizza i numeri da 1 a 20, ma succede qualcosa dopo il 2.
29
30 ##########################################################################
31
32 # Stesso ciclo, ma sostituendo 'continue' con 'break'.
33
34 a=0
35
36 while [ "$a" -le "$LIMITE" ]
37 do
38  a=$(($a+1))
39
40  if [ "$a" -gt 2 ]
41  then
42    break  # Salta l'intero ciclo.
43  fi
44
45  echo -n "$a "
46 done
47
48 echo; echo; echo
49
50 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*.

Esempio 10-21. Interrompere un ciclo ad un determinato livello

```
 1 #!/bin/bash
 2 # break-levels.sh: Interruzione di cicli.
 3
 4 # "break N" interrompe i cicli al livello N.
```

```
 5
 6 for cicloesterno in 1 2 3 4 5
 7 do
 8   echo -n "Gruppo $cicloesterno:   "
 9
10   for ciclointerno in 1 2 3 4 5
11   do
12     echo -n "$ciclointerno "
13
14     if [ "$ciclointerno" -eq 3 ]
15     then
16       break  # Provate   break 2   per vedere il risultato.
17              # ("Interrompe" entrambi i cicli, interno ed esterno).
18     fi
19   done
20
21   echo
22 done
23
24 echo
25
26 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.

Esempio 10-22. Proseguire ad un livello di ciclo superiore

```
 1 #!/bin/bash
 2 # Il comando "continue N", continua all'Nsimo livello.
 3
 4 for esterno in I II III IV V           # ciclo esterno
 5 do
 6   echo; echo -n "Gruppo $esterno: "
 7
 8   for interno in 1 2 3 4 5 6 7 8 9 10  # ciclo interno
 9   do
10
11     if [ "$interno" -eq 7 ]
12     then
13       continue 2  #  Continua al ciclo di 2° livello, cioè il
14                   #+ "ciclo esterno". Modificate la riga precedente
15                   #+ con un semplice "continue" per vedere il
16                   #+ consueto comportamento del ciclo.
17     fi
18
19     echo -n "$interno "  # 7 8 9 10 non verranno mai visualizzati.
20   done
21
22 done
23
24 echo; echo
25
26 # Esercizio:
27 # Trovate un valido uso di "continue N" in uno script.
28
29 exit 0
```

Esempio 10-23. Uso di "continue N" in un caso reale

```
  1 # Albert Reiner fornisce un esempio di come usare "continue N":
  2 # ------------------------------------------------------
  3
  4 #  Supponiamo di avere un numero elevato di job che devono essere
  5 #+ eseguiti, con tutti i dati che devono essere trattati contenuti
in un
  6 #+ file, che ha un certo nome ed è inserito in una data directory.
  7 #+ Ci sono diverse macchine che hanno accesso a questa directory e
voglio
  8 #+ distribuire il lavoro su tutte queste macchine. Per far questo,
  9 #+ solitamente, utilizzo nohup con il codice seguente su ogni
macchina:
 10
 11 while true
 12 do
 13   for n in .iso.*
 14   do
 15     [ "$n" = ".iso.opts" ] && continue
 16     beta=${n#.iso.}
 17     [ -r .Iso.$beta ] && continue
 18     [ -r .lock.$beta ] && sleep 10 && continue
 19     lockfile -r0 .lock.$beta || continue
 20     echo -n "$beta: " `date`
 21     run-isotherm $beta
 22     date
 23     ls -alF .Iso.$beta
 24     [ -r .Iso.$beta ] && rm -f .lock.$beta
 25     continue 2
 26   done
 27   break
 28 done
 29
 30 #  I dettagli, in particolare sleep N, sono specifici per la mia
 31 #+ applicazione, ma la struttura generale è:
 32
 33 while true
 34 do
 35   for job in {modello}
 36   do
 37     {job già terminati o in esecuzione} && continue
 38     {marca il job come in esecuzione, lo esegue, lo marca come
eseguito}
 39     continue 2
 40   done
 41   break        # Oppure `sleep 600' per evitare la conclusione.
 42 done
 43
 44 #  In questo modo lo script si interromperà solo quando non ci
saranno
 45 #+ più job da eseguire (compresi i job che sono stati aggiunti
durante
 46 #+ il runtime). Tramite l'uso di appropriati lockfile può essere
 47 #+ eseguito su diverse macchine concorrenti senza duplicazione di
 48 #+ calcoli [che, nel mio caso, occupano un paio d'ore, quindi è
 49 #+ veramente il caso di evitarlo]. Inoltre, poiché la ricerca
 50 #+ ricomincia sempre dall'inizio, è possibile codificare le priorità
 51 #+ nei nomi dei file. Naturalmente, questo si potrebbe fare senza
 52 #+ `continue 2', ma allora si dovrebbe verificare effettivamente se
 53 #+ alcuni job sono stati eseguiti (in questo caso dovremmo cercare
 54 #+ immediatamente il job successivo) o meno (in quest'altro dovremmo
```

```
55 #+ interrompere o sospendere l'esecuzione per molto tempo prima di
56 #+ poter verificare un nuovo job).
```


Il costrutto **continue N** è difficile da capire e complicato da usare, in modo significativo, in qualsiasi contesto. Sarebbe meglio evitarlo.

Note

[1] Sono builtin di shell, mentre altri comandi di ciclo, come while e case, sono parole chiave.

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 per creare menu.

case "$*variabile*" in

"$*condizione1*")
comando...
;;

"$*condizione2*")
comando...
;;

esac

- 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).

Esempio 10-24. Utilizzare case

```
 1 #!/bin/bash
 2 # Verificare intervalli di caratteri.
 3
```

```
 4 echo; echo "Premi un tasto e poi invio."
 5 read Tasto
 6
 7 case "$Tasto" in
 8   [[:lower:]]   ) echo "Lettera minuscola";;
 9   [[:upper:]]   ) echo "Lettera maiuscola";;
10   [0-9]         ) echo "Cifra";;
11   *             ) echo "Punteggiatura, spaziatura, o altro";;
12 esac      #  Sono permessi gli intervalli di caratteri se
13           #+ compresi tra [parentesi quadre]
14           #+ o nel formato POSIX tra [[doppie parentesi quadre.
15
16 #  La prima versione di quest'esempio usava, per indicare
17 #+ gli intervalli di caratteri minuscoli e maiuscoli, le forme
18 #+ [a-z] e [A-Z].
19 #  Questo non è più possibile per particolari impostazioni locali
20 #+ e/o distribuzioni Linux.
21 #  Grazie a Frank Wang per averlo evidenziato.
22
23 # Esercizio:
24 # --------
25 #  Così com'è, lo script accetta la pressione di un solo tasto,
quindi
26 #+ termina. Modificate lo script in modo che accetti un input
continuo,
27 #+ visualizzi ogni tasto premuto e termini solo quando viene
digitata una "X".
28 #  Suggerimento: racchiudete tutto in un ciclo "while".
29
30 exit 0
```

Esempio 10-25. Creare menu utilizzando case

```
 1 #!/bin/bash
 2
 3 # Un database di indirizzi non molto elegante
 4
 5 clear # Pulisce lo schermo.
 6
 7 echo "          Elenco Contatti"
 8 echo "          ------ --------"
 9 echo "Scegliete una delle persone seguenti:"
10 echo
11 echo "[E]vans, Roland"
12 echo "[J]ones, Mildred"
13 echo "[S]mith, Julie"
14 echo "[Z]ane, Morris"
15 echo
16
17 read persona
18
19 case "$persona" in
20 # Notate l'uso del "quoting" per la variabile.
21
22   "E" | "e" )
23   # Accetta sia una lettera maiuscola che minuscola.
24   echo
25   echo "Roland Evans"
26   echo "4321 Floppy Dr."
27   echo "Hardscrabble, CO 80753"
28   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
```

Esempio 10-26. Usare la sostituzione di comando per creare la variabile di case

```
  1 #!/bin/bash
  2 # Usare la sostituzione di comando per creare la variabile di
"case".
  3
  4 case $( arch ) in   # "arch" restituisce l'architettura della
macchina.
  5 i386 ) echo "Macchina con processore 80386";;
  6 i486 ) echo "Macchina con processore 80486";;
  7 i586 ) echo "Macchina con processore Pentium";;
  8 i686 ) echo "Macchina con processore Pentium2+";;
  9 *    ) echo "Altro tipo di macchina";;
 10 esac
 11
 12 exit 0
```

Un costrutto **case** può filtrare le stringhe in una ricerca che fa uso del globbing.

Esempio 10-27. Una semplice ricerca di stringa

```
  1 #!/bin/bash
  2 # match-string.sh: semplice ricerca di stringa
  3
  4 verifica_stringa ()
  5 {
  6   UGUALE=0
  7   NONUGUALE=90
  8   PARAM=2     # La funzione richiede 2 argomenti.
  9   ERR_PARAM=91
 10
 11   [ $# -eq $PARAM ] || return $ERR_PARAM
 12
 13   case "$1" in
 14   "$2") return $UGUALE;;
 15   *   ) return $NONUGUALE;;
 16   esac
 17
 18 }
 19
 20
 21 a=uno
 22 b=due
 23 c=tre
 24 d=due
 25
 26
 27 verifica_stringa $a     # numero di parametri errato
 28 echo $?                # 91
 29
 30 verifica_stringa $a $b  # diverse
 31 echo $?                # 90
 32
 33 verifica_stringa $b $d  # uguali
 34 echo $?                # 0
 35
 36
 37 exit 0
```

Esempio 10-28. Verificare un input alfabetico

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

```
  2 # isalpha.sh: Utilizzare la struttura "case" per filtrare una
stringa.
  3
  4 SUCCESSO=0
  5 FALLIMENTO=-1
  6
  7 isalpha ()  #  Verifica se il *primo carattere* della stringa
  8             #+ di input è una lettera.
  9 {
 10 if [ -z "$1" ]               # Nessun argomento passato?
 11 then
 12   return $FALLIMENTO
 13 fi
 14
 15 case "$1" in
 16 [a-zA-Z]*) return $SUCCESSO;;  # Inizia con una lettera?
 17 *        ) return $FALLIMENTO;;
 18 esac
 19 }             # Confrontatelo con la funzione "isalpha ()" del C.
 20
 21
 22 isalpha2 ()   # Verifica se l'*intera stringa* è composta da
lettere.
 23 {
 24   [ $# -eq 1 ] || return $FALLIMENTO
 25
 26   case $1 in
 27   *[!a-zA-Z]*|"") return $FALLIMENTO;;
 28                *) return $SUCCESSO;;
 29   esac
 30 }
 31
 32 isdigit ()    # Verifica se l'*intera stringa* è formata da cifre.
 33 {             # In altre parole, verifica se è una variabile
numerica.
 34   [ $# -eq 1 ] || return $FALLIMENTO
 35
 36   case $1 in
 37   *[!0-9]*|"") return $FALLIMENTO;;
 38              *) return $SUCCESSO;;
 39   esac
 40 }
 41
 42
 43 verifica_var ()  # Front-end per isalpha ().
 44 {
 45 if isalpha "$@"
 46 then
 47   echo "\"$*\" inizia con un carattere alfabetico."
 48   if isalpha2 "$@"
 49   then     # Non ha significato se il primo carattere non è
alfabetico.
 50     echo "\"$*\" contiene solo lettere."
 51   else
 52     echo "\"$*\" contiene almeno un carattere non alfabetico."
 53   fi
 54 else
 55   echo "\"$*\"  non inizia con una lettera."
 56                  # Stessa risposta se non viene passato alcun
argomento.
 57 fi
 58
```

```
 59 echo
 60
 61 }
 62
 63 verifica_cifra ()# Front-end per isdigit ().
 64 {
 65 if isdigit "$@"
 66 then
 67   echo "\"$*\" contiene solo cifre [0 - 9]."
 68 else
 69   echo "\"$*\" contiene almeno un carattere diverso da una cifra."
 70 fi
 71
 72 echo
 73
 74 }
 75
 76 a=23skidoo
 77 b=H3llo
 78 c=-Cosa?
 79 d=Cosa?
 80 e=`echo $b`      # Sostituzione di comando.
 81 f=AbcDef
 82 g=27234
 83 h=27a34
 84 i=27.34
 85
 86 verifica_var $a
 87 verifica_var $b
 88 verifica_var $c
 89 verifica_var $d
 90 verifica_var $e
 91 verifica_var $f
 92 verifica_var     # Non viene passato nessun argomento, cosa succede?
 93 #
 94 verifica_cifra $g
 95 verifica_cifra $h
 96 verifica_cifra $i
 97
 98
 99 exit 0           # Script perfezionato da S.C.
100
101 # Esercizio:
102 # --------
103 #  Scrivete la funzione 'isfloat ()' che verifichi i numeri in
virgola
104 #+ mobile. Suggerimento: la funzione è uguale a 'isdigit ()', ma con
105 #+ 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.

Esempio 10-29. Creare menu utilizzando select

```
  1 #!/bin/bash
  2
  3 PS3='Scegli il tuo ortaggio preferito: '# Imposta la stringa del
prompt.
  4
  5 echo
  6
  7 select verdura in "fagioli" "carote" "patate" "cipolle" "rape"
  8 do
  9   echo
 10   echo "Il tuo ortaggio preferito sono i/le $verdura."
 11   echo "Yuck!"
 12   echo
 13   break  # Cosa succederebbe se non ci fosse il "break"?
 14 done
 15
 16 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.

Esempio 10-30. Creare menu utilizzando select in una funzione

```
  1 #!/bin/bash
  2
  3 PS3='Scegli il tuo ortaggio preferito: '
  4
  5 echo
  6
  7 scelta_di()
  8 {
  9 select verdura
 10 # [in lista] omesso, quindi 'select' usa gli argomenti passati alla
funzione.
 11 do
 12   echo
 13   echo "Il tuo ortaggio preferito: $verdura."
 14   echo "Yuck!"
 15   echo
 16   break
 17 done
 18 }
 19
 20 scelta_di fagioli riso carote ravanelli pomodori spinaci
 21 #         $1      $2   $3     $4        $5       $6
 22 #         passati alla funzione scelta_di()
```

```
23
24 exit 0
```

Vedi anche Esempio 35-3.

Capitolo 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 origine *(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.

```
1 #!/bin/bash
2
3 echo "Questa riga usa il builtin \"echo\"."
4 /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 Esempio 4-1).

```
1 echo Ciao
2 echo $a
```

echo richiede l'opzione `-e` per visualizzare le sequenze di escape. Vedi Esempio 5-2.

Normalmente, ogni comando **echo** visualizza una nuova riga. L'opzione `-n` annulla questo comportamento.

echo può essere utilizzato per fornire una sequenza di comandi in una pipe.

```
 1 if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
 2 then
 3   echo "$VAR contiene la sottostringa \"txt\""
 4 fi
```

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 Esempio 12-18, Esempio 12-3, Esempio 12-38 ed Esempio 12-39.

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, comprende \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
```

Quindi, in che modo si può inserire un "a capo" in una stringa di caratteri da *visualizzare*?

```
 1 # Incorporare un a capo?
 2 echo "Perchè questa stringa non viene \n suddivisa su due righe?"
 3 # Non viene divisa.
 4
 5 # Proviamo qualcos'altro.
 6
 7 echo
 8
 9 echo $"Riga di testo contenente
10 un a capo."
11 # Viene visualizzata su due righe distinte (a capo incorporato).
12 # Ma, il prefisso di variabile "$" è proprio necessario?
13
14 echo
15
16 echo "Questa stringa è divisa
17 su due righe."
18 # No, il "$" non è necessario.
19
20 echo
21 echo "---------------"
22 echo
```

```
23
24 echo -n $"Un'altra riga di testo contenente
25 un a capo."
26 # Viene visualizzata su due righe (a capo incorporato).
27 # In questo caso neanche l'opzione -n riesce a sopprimere l'a capo.
28
29 echo
30 echo
31 echo "---------------"
32 echo
33 echo
34
35 # Tuttavia, quello che segue non funziona come potremmo aspettarci.
36 # Perchè?
37 stringa1=$"Ancora un'altra riga di testo contenente
38 un a capo (forse)."
39
40 echo $stringa1
41
42 # Grazie a Steve Parker per la precisazione.
```

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).

Le versioni più vecchie di Bash potrebbero non supportare **printf**.

Esempio 11-1. printf in azione

```
 1 #!/bin/bash
 2 # printf demo
 3
 4 PI=3,14159265358979                          # Vedi nota a fine
listato
 5 CostanteDecimale=31373
 6 Messaggio1="Saluti,"
 7 Messaggio2="un abitante della Terra."
 8
 9 echo
10
11 printf "Pi con 2 cifre decimali = %1.2f" $PI
12 echo
13 printf "Pi con 9 cifre decimali = %1.9f" $PI  #  Esegue anche il
```

```
corretto
 14                                                #+ arrotondamento.
 15
 16 printf "\n"                                    #  Esegue un ritorno a
capo,
 17                                                #  equivale a 'echo'.
 18
 19 printf "Costante = \t%d\n" $CostanteDecimale  #  Inserisce un
carattere
 20                                                #+ di tabulazione (\t)
 21
 22 printf "%s %s \n" $Messaggio1 $Messaggio2
 23
 24 echo
 25
 26 # ================================================#
 27 # Simulazione della funzione 'sprintf' del C.
 28 # Impostare una variabile con una stringa di formato.
 29
 30 echo
 31
 32 Pi12=$(printf "%1.12f" $PI)
 33 echo "Pi con 12 cifre decimali = $Pi12"
 34
 35 Msg=`printf "%s %s \n" $Messaggio1 $Messaggio2`
 36 echo $Msg; echo $Msg
 37
 38 # Ora possiamo disporre della funzione 'sprintf' come modulo
 39 # caricabile per Bash. Questo, però, non è portabile.
 40
 41 exit 0
 42
 43 #  N.d.T. Nella versione originale veniva usato il punto come
separatore
 44 #+ decimale. Con le impostazioni locali italiane il punto avrebbe
 45 #+ impedito il corretto funzionamento di printf.
```

Un'utile applicazione di **printf** è quella di impaginare i messaggi d'errore

```
  1 E_ERR_DIR=65
  2
  3 var=directory_inesistente
  4
  5 errore()
  6 {
  7   printf "$@" >&2
  8   # Organizza i parametri posizionali passati e li invia allo
stderr.
  9   echo
 10   exit $E_ERR_DIR
 11 }
 12
 13 cd $var || errore $"Non riesco a cambiare in %s." "$var"
 14
 15 # 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 Esempio 26-6).

Esempio 11-2. Assegnamento di variabile utilizzando read

```
 1 #!/bin/bash
 2
 3 echo -n "Immetti il valore della variabile 'var1': "
 4 # L'opzione -n di echo sopprime il ritorno a capo.
 5
 6 read var1
 7 #  Notate che non vi è nessun '$' davanti a var1, perché la
variabile
 8 #+ è in fase di impostazione.
 9
10 echo "var1 = $var1"
11
12
13 echo
14
15 # Un singolo enunciato 'read' può impostare più variabili.
16 echo -n "Immetti i valori delle variabili 'var2' e 'var3' (separati
da \
17 uno spazio o da tab): "
18 read var2 var3
19 echo "var2 = $var2      var3 = $var3"
20 #  Se si immette un solo valore, le rimanenti variabili restano non
21 #+ impostate (nulle).
22
23 exit 0
```

read, senza una variabile associata, assegna l'input alla variabile dedicata $REPLY.

Esempio 11-3. Cosa succede quando read non è associato ad una variabile

```
 1 #!/bin/bash
 2 # read-novar.sh
 3
 4 echo
 5
 6 # -------------------------- #
 7 echo -n "Immetti un valore: "
 8 read var
 9 echo "\"var\" = "$var""
10 # Tutto come ci si aspetta.
11 # -------------------------- #
12
13 echo
14
15 # ------------------------------------------------------------- #
16 echo -n "Immetti un altro valore: "
17 read           #  Non viene fornita alcuna variabile a 'read',
18               #+ quindi... l'input di 'read' viene assegnato alla
19               #+ variabile predefinita $REPLY.
20 var="$REPLY"
21 echo "\"var\" = "$var""
22 # Stesso risultato del primo blocco di codice.
23 # ------------------------------------------------------------- #
24
25 echo
26
27 exit 0
```

Normalmente, immettendo una \ nell'input di **read** si disabilita il ritorno a capo. L'opzione `-r` consente di interpretare la \ letteralmente.

Esempio 11-4. Input su più righe per read

```
  1 #!/bin/bash
  2
  3 echo
  4
  5 echo "Immettete una stringa che termina con \\, quindi premete
<INVIO>."
  6 echo "Dopo di che, immettete una seconda stringa e premete ancora
<INVIO>."
  7 read var1     # La "\" sopprime il ritorno a capo durante la lettura
di "var1".
  8               #     prima riga \
  9               #     seconda riga
 10
 11 echo "var1 = $var1"
 12 #     var1 = prima riga seconda riga
 13
 14 #  Per ciascuna riga che termina con "\", si ottiene un prompt alla
riga
 15 #+ successiva per continuare ad inserire caratteri in var1.
 16
 17 echo; echo
 18
 19 echo "Immettete un'altra stringa che termina con \\ , quindi premete
<INVIO>."
 20 read -r var2  # L'opzione -r fa sì che "\" venga interpretata
letteralmente.
 21              #     prima riga \
 22
 23 echo "var2 = $var2"
 24 #     var2 = prima riga \
 25
 26 # L'introduzione dei dati termina con il primo <INVIO>.
 27
 28 echo
 29
 30 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**.

```
  1 # Rilevare la pressione di un tasto senza dover premere INVIO.
  2
  3 read -s -n1 -p "Premi un tasto " tasto
  4 echo; echo "Hai premuto il tasto "\"$tasto\""."
  5
  6 #  L'opzione -s serve a non visualizzare l'input.
  7 #  L'opzione -n N indica che devono essere accettati solo N
caratteri di input.
  8 #  L'opzione -p permette di visualizzare il messaggio del prompt
immediatamente
  9 #+ successivo, prima di leggere l'input.
 10
 11 #  Usare queste opzioni è un po' complicato, perché
 12 #+ devono essere poste nell'ordine esatto.
```

L'opzione `-n` di **read** consente anche il rilevamento dei *tasti freccia* ed alcuni altri tasti inusuali.

Esempio 11-5. Rilevare i tasti freccia

```
 1 #!/bin/bash
 2 # arrow-detect.sh: Rileva i tasti freccia, e qualcos'altro.
 3 # Grazie a Sandro Magi per avermelo mostrato.
 4
 5 # --------------------------------------------------------
 6 # Codice dei caratteri generati dalla pressione dei tasti.
 7 frecciasu='\[A'
 8 frecciagiù='\[B'
 9 frecciadestra='\[C'
10 frecciasinistra='\[D'
11 ins='\[2'
12 canc='\[3'
13 # --------------------------------------------------------
14
15 SUCCESSO=0
16 ALTRO=65
17
18 echo -n "Premi un tasto...  "
19 #  Potrebbe essere necessario premere anche INVIO se viene digitato
un
20 #+ tasto non tra quelli elencati.
21 read -n3 tasto                          # Legge 3 caratteri.
22
23 echo -n "$tasto" | grep "$frecciasu"  #  Verifica il codice del
24                                        #+ tasto premuto.
25 if [ "$?" -eq $SUCCESSO ]
26 then
27   echo "È stato premuto il tasto Freccia-su."
28   exit $SUCCESSO
29 fi
30
31 echo -n "$tasto" | grep "$frecciagiù"
32 if [ "$?" -eq $SUCCESSO ]
33 then
34   echo "È stato premuto il tasto Freccia-giù."
35   exit $SUCCESSO
36 fi
37
38 echo -n "$tasto" | grep "$frecciadestra"
39 if [ "$?" -eq $SUCCESSO ]
40 then
41   echo "È stato premuto il tasto Freccia-destra."
42   exit $SUCCESSO
43 fi
44
45 echo -n "$tasto" | grep "$frecciasinistra"
46 if [ "$?" -eq $SUCCESSO ]
47 then
48   echo "È stato premuto il tasto Freccia-sinistra."
49   exit $SUCCESSO
50 fi
51
52 echo -n "$tasto" | grep "$ins"
53 if [ "$?" -eq $SUCCESSO ]
54 then
55   echo "È stato premuto il tasto \"Ins\"."
```

```
 56   exit $SUCCESSO
 57 fi
 58
 59 echo -n "$tasto" | grep "$canc"
 60 if [ "$?" -eq $SUCCESSO ]
 61 then
 62   echo "È stato premuto il tasto \"Canc\"."
 63   exit $SUCCESSO
 64 fi
 65
 66
 67 echo " È stato premuto un altro tasto."
 68
 69 exit $ALTRO
 70
 71 #  Esercizi:
 72 #  ---------
 73 #  1) Semplificate lo script trasformando le verifiche multiple "if"
in un
 74 #     costrutto 'case'.
 75 #  2) Aggiungete il rilevamento dei tasti "Home", "Fine", "PgUp" e
"PgDn".
 76
 77 #  N.d.T. Attenzione! I codici dei tasti indicati all'inizio
potrebbero non
 78 #+ corrispondere a quelli della vostra tastiera.
 79 #  Verificateli e quindi, modificate l'esercizio in modo che
funzioni
 80 #+ correttamente.
```


L'opzione `-n` di **read** evita il rilevamento del tasto **INVIO** (nuova riga).

L'opzione `-t` di **read** consente un input temporizzato (vedi Esempio 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!

Esempio 11-6. Utilizzare read con la redirezione di file

```
  1 #!/bin/bash
  2
  3 read var1 <file-dati
  4 echo "var1 = $var1"
  5 # var1 viene impostata con l'intera prima riga del file di input
"file-dati"
  6
  7 read var2 var3 <file-dati
  8 echo "var2 = $var2   var3 = $var3"
  9 # Notate qui il comportamento poco intuitivo di "read".
 10 # 1) Ritorna all'inizio del file di input.
 11 # 2) Ciascuna variabile viene impostata alla stringa corrispondente,
 12 #    separata da spazi, piuttosto che all'intera riga di testo.
 13 # 3) La variabile finale viene impostata alla parte rimanente della
riga.
 14 # 4) Se ci sono più variabili da impostare di quante siano le
 15 #    stringhe separate da spazi nella prima riga del file, allora le
 16 #    variabili in eccesso restano vuote.
 17
```

```
 18 echo "-----------------------------------------------"
 19
 20 # Come risolvere il problema precedente con un ciclo:
 21 while read riga
 22 do
 23   echo "$riga"
 24 done <file-dati
 25 # Grazie a Heiner Steven per la puntualizzazione.
 26
 27 echo "-----------------------------------------------"
 28
 29 #  Uso della variabile $IFS (Internal Field Separator) per
suddividere
 30 #+ una riga di input per "read",
 31 #+ se non si vuole che il delimitatore preimpostato sia la
spaziatura.
 32
 33 echo "Elenco di tutti gli utenti:"
 34 OIFS=$IFS; IFS=:       # /etc/passwd usa ":" come separatore di
campo.
 35 while read name passwd uid gid fullname ignore
 36 do
 37   echo "$name ($fullname)"
 38 done <etc/passwd       # Redirezione I/O.
 39 IFS=$OIFS              # Ripristina il valore originario di $IFS.
 40 # Anche questo frammento di codice è di Heiner Steven.
 41
 42
 43
 44 #  Impostando la variabile $IFS all'interno dello stesso ciclo,
 45 #+ viene eliminata la necessità di salvare il valore originario
 46 #+ di $IFS in una variabile temporanea.
 47 #  Grazie, Dim Segebart per la precisazione.
 48 echo "-----------------------------------------------"
 49 echo "Elenco di tutti gli utenti:"
 50
 51 while IFS=: read name passwd uid gid fullname ignore
 52 do
 53   echo "$name ($fullname)"
 54 done <etc/passwd       # Redirezione I/O.
 55
 56 echo
 57 echo "\$IFS è ancora $IFS"
 58
 59 exit 0
```

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.

```
  1 cat file1 file2 |
  2 while read riga
  3 do
  4 echo $riga
  5 done
```

Comunque, come mostra Björn Eriksson:

Esempio 11-7. Problemi leggendo da una pipe

```
 1 #!/bin/sh
 2 # readpipe.sh
 3 # Esempio fornito da Bjon Eriksson.
 4
 5 ultimo="(null)"
 6 cat $0 |
 7 while read riga
 8 do
 9     echo "{$riga}"
10     ultimo=$riga
11 done
12 printf "\nFatto, ultimo:$ultimo\n"
13
14 exit 0  # Fine del codice.
15         # Segue l'output (parziale) dello script.
16         # 'echo' fornisce le parentesi graffe aggiuntive.
17
18 ############################################################
19
20 ./readpipe.sh
21
22 {#!/bin/sh}
23 {ultimo="(null)"}
24 {cat $0 |}
25 {while read riga}
26 {do}
27 {echo "{$riga}"}
28 {ultimo=$riga}
29 {done}
30 {printf "nFatto, ultimo:$ultimon"}
31
32
33 Fatto, ultimo:(null)
34
35 La variabile (ultimo) è stata impostata all'interno di
una subshell,
36 al di fuori di essa, quindi, rimane non impostata.
```

Lo script **gendiff**, che di solito si trova in `/usr/bin` in molte distribuzioni Linux, usa una pipe per collegare l'output di find ad un costrutto *while read*.

```
1 find $1 \( -name "*$2" -o -name ".*$2" \) -print |
2 while read f; do
3 . . .
```

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.

```
 1 (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.


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 Esempio 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.

Esempio 11-8. Cambiare la directory di lavoro corrente

```
 1 #!/bin/bash
 2
 3 dir1=/usr/local
 4 dir2=/var/spool
 5
 6 pushd $dir1
 7 #  Viene eseguito un 'dirs' automatico (visualizza lo stack delle
 8 #+ directory allo stdout).
```

```
 9 echo "Ora sei nella directory `pwd`." #  Uso degli apici singoli
10                                       #+ inversi per 'pwd'.
11
12 # Ora si fa qualcosa nella directory 'dir1'.
13 pushd $dir2
14 echo "Ora sei nella directory `pwd`."
15
16 # Adesso si fa qualcos'altro nella directory 'dir2'.
17 echo "Nella posizione più alta dell'array DIRSTACK si trova
$DIRSTACK."
18 popd
19 echo "Sei ritornato alla directory `pwd`."
20
21 # Ora si fa qualche altra cosa nella directory 'dir1'.
22 popd
23 echo "Sei tornato alla directory di lavoro originaria `pwd`."
24
25 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.

Esempio 11-9. Facciamo fare a "let" qualche calcolo aritmetico.

```
 1 #!/bin/bash
 2
 3 echo
 4
 5 let a=11            # Uguale a 'a=11'
 6 let a=a+5           # Equivale a  let "a = a + 5"
 7                     # (i doppi apici e gli spazi la rendono più
leggibile.)
 8 echo "11 + 5 = $a"  # 16
 9
10 let "a <<= 3"       # Equivale a  let "a = a << 3"
11 echo "\"\$a\" (=16) scorrimento a sinistra di 3 bit = $a"
12                     # 128
13
14 let "a /= 4"        # Equivale a  let "a = a / 4"
15 echo "128 / 4 = $a" # 32
16
17 let "a -= 5"        # Equivale a  let "a = a - 5"
18 echo "32 - 5 = $a"  # 27
19
20 let "a = a * 10"    # Equivale a  let "a = a * 10"
21 echo "27 * 10 = $a" # 270
22
23 let "a %= 8"        # Equivale a  let "a = a % 8"
24 echo "270 modulo 8 = $a  (270 / 8 = 33, resto $a)"
25                     # 6
26
27 echo
28
29 exit 0
```

eval

eval arg1 [arg2] ... [argN]

Combina gli argomenti presenti in un'espressione, o in una lista di espressioni, e li *valuta*. Espande qualsiasi variabile presente nell'espressione. Il resultato viene tradotto in un comando. Può essere utile per generare del codice da riga di comando o dentro uno script.

```
bash$ processo=xterm
bash$ mostra_processo="eval ps ax | grep $processo"
bash$ $mostra_processo
1867 tty1     S      0:02 xterm
2779 tty1     S      0:00 xterm
2886 pts/1    S      0:00 grep xterm
```

Esempio 11-10. Dimostrazione degli effetti di eval

```
  1 #!/bin/bash
  2
  3 y=`eval ls -l`  #  Simile a  y=`ls -l`
  4 echo $y         #+ ma con i ritorni a capo tolti perché la variabile
  5                 #+ "visualizzata" è senza "quoting".
  6 echo
  7 echo "$y"       #  I ritorni a capo vengono mantenuti con il
  8                 #+ "quoting" della variabile.
  9
 10 echo; echo
 11
 12 y=`eval df`     #  Simile a y=`df`
 13 echo $y         #+ ma senza ritorni a capo.
 14
 15 #  Se non si preservano i ritorni a capo, la verifica dell'output
 16 #+ con utility come "awk" risulta più facile.
 17
 18 echo
 19 echo
"============================================================================"
 20 echo
 21
 22 # Ora vediamo come "espandere" una variabile usando "eval" . . .
 23
 24 for i in 1 2 3 4 5; do
 25   eval valore=$i
 26   #  valore=$i ha lo stesso effetto. "eval", in questo caso, non è
necessario.
 27   #  Una variabile senza meta-significato valuta se stessa --
 28   #+ non può espandersi a nient'altro che al proprio contenuto
letterale.
 29   echo $valore
 30 done
 31
 32 echo
 33 echo "---"
 34 echo
 35
 36 for i in ls df; do
 37   valore=eval $i
 38   #  valore=$i in questo caso avrebbe un effetto completamente
diverso.
 39   #  "eval" valuta i comandi "ls" e "df" . . .
 40   #  I termini "ls" e "df" hanno un meta-significato,
```

```
 41   #+ dal momento che sono interpretati come comandi
 42   #+ e non come stringhe di caratteri.
 43   echo $valore
 44 done
 45
 46
 47 exit 0
```

Esempio 11-11. Forzare un log-off

```
  1 #!/bin/bash
  2 Terminare ppp per forzare uno scollegamento.
  3
  4 Lo script deve essere eseguito da root.
  5
  6 terminappp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
  7 #                        ----- ID di processo di ppp ------
  8
  9 $terminappp             # La variabile è diventata un comando.
 10
 11
 12 # Le operazioni seguenti devono essere eseguite da root.
 13
 14 chmod 666 /dev/ttyS3    # Occorrono i permessi di lettura+scrittura,
altrimenti?
 15 #  Quando si invia un SIGKILL a ppp i permessi della porta seriale
vengono
 16 #+ modificati, quindi vanno ripristinati allo stato precedente il
SIGKILL.
 17
 18 rm /var/lock/LCK..ttyS3 # Cancella il lock file della porta seriale.
Perchè?
 19
 20 exit 0
 21
 22 # Esercizi:
 23 # --------
 24 # 1) Lo script deve verificare se è stato root ad invocarlo.
 25 # 2) Effettuate un controllo per verificare che, prima di tentarne
la chiusura,
 26 #+   il processo che deve essere terminato sia effettivamente in
esecuzione.
```

Esempio 11-12. Una versione di "rot13"

```
  1 #!/bin/bash
  2 # Una versione di "rot13" usando 'eval'.
  3 # Confrontatelo con l'esempio "rot13.sh".
  4
  5 impvar_rot_13()              # Codifica "rot13"
  6 {
  7   local nomevar=$1 valoreval=$2
  8   eval $nomevar='$(echo "$valoreval" | tr a-z n-za-m)'
  9 }
 10
 11
 12 impvar_rot_13 var "foobar"   # Codifica "foobar" con rot13.
 13 echo $var                   # sbbone
 14
 15 echo $var | tr a-z n-za-m    # Codifica "sbbone" con rot13.
```

```
 16                              # Ritorno al valore originario della
variabile.
 17 echo $var                    # foobar
 18
 19 # Esempio di Stephane Chazelas.
 20 # Modificato dall'autore del documento.
 21
 22 exit 0
```

Rory Winston ha fornito il seguente esempio che dimostra quanto possa essere utile **eval**.

Esempio 11-13. Utilizzare eval per forzare una sostituzione di variabile in uno script Perl

```
  1 Nello script Perl  "test.pl":
  2         ...
  3         my $WEBROOT = <WEBROOT_PATH>;
  4         ...
  5
  6 Per forzare la sostituzione di variabile provate:
  7         $export WEBROOT_PATH=/usr/local/webroot
  8         $sed 's/<WEBROOT_PATH>/$WEBROOT_PATH/' < test.pl > out
  9
 10 Ma questo dà solamente:
 11         my $WEBROOT = $WEBROOT_PATH;
 12
 13 Tuttavia:
 14         $export WEBROOT_PATH=/usr/local/webroot
 15         $eval sed 's%\<WEBROOT_PATH\>%$WEBROOT_PATH%' < test.pl >
out
 16 #        ====
 17
 18 Che funziona bene, eseguendo l'attesa sostituzione:
 19         my $WEBROOT = /usr/local/webroot;
 20
 21
 22 ### Correzioni all'esempio originale eseguite da Paulo Marcel Coelho
Aragao.
 23
```

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.

Esempio 11-14. Utilizzare set con i parametri posizionali

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

```
 3 # script "set-test"
 4
 5 # Invocate lo script con tre argomenti da riga di comando,
 6 # per esempio, "./set-test uno due tre".
 7
 8 echo
 9 echo "Parametri posizionali prima di  set \`uname -a\` :"
10 echo "Argomento nr.1 da riga di comando = $1"
11 echo "Argomento nr.2 da riga di comando = $2"
12 echo "Argomento nr.3 da riga di comando = $3"
13
14
15 set `uname -a`  # Imposta i parametri posizionali all'output
16                 # del comando `uname -a`
17
18 echo $_         # Sconosciuto
19 # Opzioni impostate nello script.
20
21 echo "Parametri posizionali dopo  set \`uname -a\` :"
22 # $1, $2, $3, ecc. reinizializzati col risultato di `uname -a`
23 echo "Campo nr.1 di 'uname -a' = $1"
24 echo "Campo nr.2 di 'uname -a' = $2"
25 echo "Campo nr.3 di 'uname -a' = $3"
26 echo ---
27 echo $_         # ---
28 echo
29
30 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*.

Esempio 11-15. Riassegnare i parametri posizionali

```
 1 #!/bin/bash
 2
 3 variabile="uno due tre quattro cinque"
 4
 5 set -- $variabile
 6 # Imposta i parametri posizionali al contenuto di "$variabile".
 7
 8 primo_param=$1
 9 secondo_param=$2
10 shift; shift        # Salta i primi due parametri posizionali.
```

```
 11 restanti_param="$*"
 12
 13 echo
 14 echo "primo parametro = $primo_param"             # uno
 15 echo "secondo parametro = $secondo_param"         # due
 16 echo "rimanenti parametri = $restanti_param"      # tre quattro
cinque
 17
 18 echo; echo
 19
 20 # Ancora.
 21 set -- $variabile
 22 primo_param=$1
 23 secondo_param=$2
 24 echo "primo parametro = $primo_param"             # uno
 25 echo "secondo parametro = $secondo_param"         # due
 26
 27 # ======================================================
 28
 29 set --
 30 #  Annulla i parametri posizionali quando non viene specificata
 31 #+ nessuna variabile.
 32
 33 primo_param=$1
 34 secondo_param=$2
 35 echo "primo parametro = $primo_param"             # (valore nullo)
 36 echo "secondo parametro = $secondo_param"         # (valore nullo)
 37
 38 exit 0
```

Vedi anche Esempio 10-2 e Esempio 12-46.

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$
```

Esempio 11-16. "Annullare" una variabile

```
  1 #!/bin/bash
  2 # unset.sh: Annullare una variabile.
  3
  4 variabile=ciao                              # Inizializzata.
  5 echo "variabile = $variabile"
  6
  7 unset variabile                             # Annullata.
  8                                             # Stesso effetto di
variabile=
  9 echo "variabile (annullata) = $variabile"  # $variabile è nulla.
 10
 11 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.

Esempio 11-17. Utilizzare export per passare una variabile ad uno script awk incorporato

```
 1 #!/bin/bash
 2
 3 #  Ancora un'altra versione dello script "totalizzatore di colonna"
 4 #+ (col-totaler.sh) che aggiunge una specifica colonna (di numeri)
 5 #+ nel file di destinazione. Viene sfruttato l'ambiente per passare
 6 #+ una variabile dello script ad 'awk'.
 7
 8 ARG=2
 9 E_ERR_ARG=65
10
11 if [ $# -ne "$ARG" ] #  Verifica il corretto numero di argomenti da
12                       #+ riga di comando.
13 then
14    echo "Utilizzo: `basename $0` nomefile colonna-numero"
15    exit $E_ERR_ARG
16 fi
17
18 nomefile=$1
19 colonna_numero=$2
20
21 #===== Fino a questo punto è uguale allo script originale =====#
22
23 export colonna_numero
24 # Esporta il numero di colonna all'ambiente, in modo che sia
disponibile
25 #+ all'utilizzo.
26
27
28 # Inizio dello script awk.
29 # ------------------------------------------------
30 awk '{ totale += $ENVIRON["colonna_numero"]
31 }
32 END { print totale }' $nomefile
33 # ------------------------------------------------
34 # Fine dello script awk.
35
36
37 # Grazie, Stephane Chazelas.
38
39 exit 0
```


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.

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 deprecato comando esterno getopt.

```
 1 while getopts ":abcde:fg" Opzione
 2 # Dichiarazione iniziale.
 3 # a, b, c, d, e, f, g sono le opzioni attese.
 4 # I : dopo l'opzione 'e' indicano che c'è un argomento associato.
 5 do
 6   case $Opzione in
 7   a ) # Fa qualcosa con la variabile 'a'.
 8   b ) # Fa qualcosa con la variabile 'b'.
 9   ...
10   e)  # Fa qualcosa con 'e', e anche con $OPTARG,
```

```
 11       # che è l'argomento associato all'opzione 'e'.
 12   ...
 13   g ) # Fa qualcosa con la variabile 'g'.
 14   esac
 15 done
 16 shift $(($OPTIND - 1))
 17 # Sposta il puntatore all'argomento successivo.
 18
 19 # Tutto questo non è affatto complicato come sembra <sorriso>.
 20
```

Esempio 11-18. Utilizzare getopts per leggere le opzioni o gli argomenti passati ad uno script

```
  1 #!/bin/bash
  2 # Prove con getopts e OPTIND
  3 # Script modificato il 9/10/03 su suggerimento di Bill Gradwohl.
  4
  5 #  Osserviamo come 'getopts' elabora gli argomenti passati allo
script da
  6 #+ riga di comando.
  7 #  Gli argomenti vengono verificati come "opzioni" (flag)
  8 #+ ed argomenti associati.
  9
 10 #  Provate ad invocare lo script con
 11 #  'nomescript -mn'
 12 #  'nomescript -oq qOpzione' (qOpzione può essere una stringa
qualsiasi.)
 13 #  'nomescript -qXXX -r'
 14 #
 15 #  'nomescript -qr'    - Risultato inaspettato, considera "r"
 16 #+ come l'argomento dell'opzione "q"
 17 #  'nomescript -q -r'  - Risultato inaspettato, come prima.
 18 #  'nomescript -mnop -mnop'  - Risultato inaspettato
 19 #  (OPTIND non è attendibile nello stabilire da dove proviene
un'opzione).
 20 #
 21 #  Se un'opzione si aspetta un argomento ("flag:"), viene presa
 22 #  qualunque cosa si trovi vicino.
 23
 24 NO_ARG=0
 25 E_ERR_OPZ=65
 26
 27 if [ $# -eq "$NO_ARG" ]   #  Lo script è stato invocato senza
 28                           #+ alcun argomento?
 29 then
 30   echo "Utilizzo: `basename $0` opzioni (-mnopqrs)"
 31   exit $E_ERR_OPZ         #  Se non ci sono argomenti, esce e
 32                           #+ spiega come usare lo script.
 33 fi
 34 # Utilizzo: nomescript -opzioni
 35 # Nota: è necessario il trattino (-)
 36
 37
 38 while getopts ":mnopq:rs" Opzione
 39 do
 40   case $Opzione in
 41     m     ) echo "Scenario nr.1: opzione -m-  [OPTIND=${OPTIND}]";;
 42     n | o ) echo "Scenario nr.2: opzione -$Opzione-
[OPTIND=${OPTIND}]";;
 43     p     ) echo "Scenario nr.3: opzione -p-  [OPTIND=${OPTIND}]";;
```

```
 44     q     ) echo "Scenario nr.4: opzione -q-\
 45 con argomento \"$OPTARG\"  [OPTIND=${OPTIND}]";;
 46     # Notate che l'opzione 'q' deve avere un argomento associato,
 47     # altrimenti salta alla voce predefinita del costrutto case.
 48     r | s ) echo "Scenario nr.5: opzione -$Opzione-"'';;
 49     *     ) echo "È stata scelta un'opzione non implementata.";; #
DEFAULT
 50   esac
 51 done
 52
 53 shift $(($OPTIND - 1))
 54 #  Decrementa il puntatore agli argomenti in modo che punti al
successivo.
 55 #  $1 fa ora riferimento al primo elemento non-opzione fornito da
riga di
 56 #+ comando, ammesso che ci sia.
 57
 58 exit 0
 59
 60 #   Come asserisce Bill Gradwohl,
 61 #  "Il funzionamento di getopts permette di specificare: nomescript
-mnop -mnop,
 62 #+  ma non esiste, utilizzando OPTIND, nessun modo affidabile per
differenziare
 63 #+  da dove proviene che cosa."
```

Comportamento dello Script

source, . (comando punto)

Questa istruzione, se invocata da riga di comando, esegue uno script. All'interno di uno script, **`source nome-file`** carica il file `nome-file`. Caricando un file (comando-punto) si *importa* codice all'interno dello script, accodandolo (stesso effetto della direttiva **`#include`** di un programma C). Il risultato finale è uguale all'"inserimento" di righe di codice nel corpo dello script. È utile in situazioni in cui diversi script usano un file dati comune o una stessa libreria di funzioni.

Esempio 11-19. "Includere" un file dati

```
  1 #!/bin/bash
  2
  3 . file-dati    # Carica un file dati.
  4 # Stesso effetto di "source file-dati", ma più portabile.
  5
  6 #  Il file "file-dati" deve essere presente nella directory di
lavoro
  7 #+ corrente, poiché vi si fa riferimento per mezzo del suo
'basename'.
  8
  9 # Ora utilizziamo alcuni dati del file.
 10
 11 echo "variabile1 (dal file-dati) = $variabile1"
 12 echo "variabile3 (dal file-dati) = $variabile3"
 13
 14 let "sommma = $variabile2 + $variabile4"
 15 echo "Somma della variabile2 + variabile4 (dal file-dati) = $somma"
 16 echo "messaggio1 (dal file-dati)  \"$messaggio1\""
 17 # Nota:                           apici doppi con escape.
 18
```

```
 19 visualizza_messaggio Questa è la funzione di visualizzazione
messaggio \
 20 presente in file-dati.
 21
 22
 23 exit 0
```

Il file `file-dati` per l'Esempio 11-19 precedente. Dev'essere presente nella stessa directory.

```
  1 #  Questo è il file dati caricato dallo script.
  2 #  File di questo tipo possono contenere variabili, funzioni, ecc.
  3 #  Può essere caricato con il comando 'source' o '.' da uno script
di shell.
  4
  5 # Inizializziamo alcune variabili.
  6
  7 variabile1=22
  8 variabile2=474
  9 variabile3=5
 10 variabile4=97
 11
 12 messaggio1="Ciao, come stai?"
 13 messaggio2="Per ora piuttosto bene. Arrivederci."
 14
 15 visualizza_messaggio ()
 16 {
 17 # Visualizza qualsiasi messaggio passato come argomento.
 18
 19   if [ -z "$1" ]
 20   then
 21   return 1
 22   # Errore, se l'argomento è assente.
 23   fi
 24
 25   echo
 26
 27   until [ -z "$1" ]
 28   do
 29   # Scorre gli argomenti passati alla funzione.
 30     echo -n "$1"
 31     #  Visualizza gli argomenti uno alla volta, eliminando i ritorni
a capo.
 32     echo -n " "
 33     # Inserisce degli spazi tra le parole.
 34     shift
 35     # Successivo.
 36   done
 37
 38   echo
 39
 40   return 0
 41 }
```

È anche possibile per uno script usare *source* in riferimento a se stesso, sebbene questo non sembri avere reali applicazioni pratiche.

Esempio 11-20. Un (inutile) script che esegue se stesso

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

```
  2 # self-source.sh: uno script che segue se stesso "ricorsivamente."
  3 # Da "Stupid Script Tricks," Volume II.
  4
  5 MAXPASSCNT=100    # Numero massimo di esecuzioni.
  6
  7 echo -n  "$conta_passi  "
  8 #  Al primo passaggio, vengono visualizzati solo due spazi,
  9 #+ perché $conta_passi non è stata inizializzata.
 10
 11 let "conta_passi += 1"
 12 #  Si assume che la variabile $conta_passi non inizializzata possa
essere
 13 #+ incrementata subito.
 14 #  Questo funziona con Bash e pdksh, ma si basa su un'azione non
portabile
 15 #+ (e perfino pericolosa).
 16 #  Sarebbe meglio impostare $conta_passi a 0 nel caso non fosse
stata
 17 #+ inizializzata.
 18
 19 while [ "$conta_passi" -le $MAXPASSCNT ]
 20 do
 21   . $0   #  Lo script "esegue" se stesso, non chiama se stesso.
 22          #  ./$0 (che sarebbe la vera ricorsività) in questo caso
non funziona.
 23 done
 24
 25 #  Quello che avviene in questo script non è una vera ricorsività,
perché lo
 26 #+ script in realtà "espande" se stesso (genera una nuova sezione di
codice)
 27 #+ ad ogni passaggio attraverso il ciclo 'while', con ogni 'source'
che si
 28 #+ trova alla riga 21.
 29 #
 30 #  Naturalmente, lo script interpreta ogni succesiva 'esecuzione'
della riga
 31 #+ con "#!" come un commento e non come l'inizio di un nuovo script.
 32
 33 echo
 34
 35 exit 0   # Il risultato finale è un conteggio da 1 a 100.
 36         # Molto impressionante.
 37
 38 # Esercizio:
 39 # --------
 40 #  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.


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.

Esempio 11-21. Effetti di exec

```
 1 #!/bin/bash
 2
 3 exec echo "Uscita da \"$0\"." # Esce dallo script in questo punto.
 4
 5 # -------------------------------------------
 6 # Le righe seguenti non verranno mai eseguite.
 7
 8 echo "Questo messaggio non verrà mai visualizzato."
 9
10 exit 99          #  Questo script non termina qui.
11                  #  Verificate l'exit status, dopo che lo script è
12                  #+ terminato, con 'echo $?'.
13                  #  *Non* sarà 99.
```

Esempio 11-22. Uno script che esegue se stesso con exec

```
 1 #!/bin/bash
 2 # self-exec.sh
 3
 4 echo
 5
 6 echo "Sebbene questa riga compaia UNA SOLA VOLTA nello script,
continuerà"
 7 echo "ad essere visualizzata."
 8 echo "Il PID di questo script d'esempio è ancora $$."
 9 #     Dimostra che non viene generata una subshell.
10
11 echo "==================== Premi Ctl-C per uscire
===================="
12
13 sleep 1
14
15 exec $0   #  Inizia un'altra istanza di questo stesso script
16           #+ che sostituisce quella precedente.
17
18 echo "Questa riga non verrà mai visualizzata"  # Perché?
19
20 exit 0
```

exec serve anche per riassegnare i descrittori dei file. **`exec <zzz-file`** sostituisce lo `stdin` con il file `zzz-file` (vedi Esempio 16-1).

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 Esempio 24-1 e Esempio 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.

```
  1 shopt -s cdspell
  2 #  Consente le errate digitazioni, non gravi, dei nomi delle
directory quando
  3 #+ si usa 'cd'
  4
  5 cd /hpme  # Oops! Errore '/home'.
  6 pwd       # /home
  7           # La shell ha corretto l'errore di digitazione.
```

Comandi

true

Comando che restituisce zero come exit status di una corretta esecuzione, ma nient'altro.

```
  1 # Ciclo infinito
  2 while true   # alternativa a ":"
  3 do
  4    operazione-1
  5    operazione-2
  6    ...
  7    operazione-n
  8    # Occorre un sistema per uscire dal ciclo, altrimenti lo script
si blocca.
  9 done
```

false

Comando che restituisce l'exit status di una esecuzione non andata a buon fine, ma nient'altro.

```
  1 # Ciclo che non viene eseguito
  2 while false
  3 do
  4    # Il codice seguente non verrà eseguito.
  5    operazione-1
  6    operazione-2
  7    ...
  8    operazione-n
  9    # Non succede niente!
 10 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.

bind

Il builtin **bind** visualizza o modifica la configurazione d'uso della tastiera tramite *readline* [3].

help

Fornisce un breve riepilogo dell'utilizzo di un builtin di shell. È il corrispettivo di whatis, 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**.

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 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).

Esempio 11-23. Attendere la fine di un processo prima di continuare

```
  1 #!/bin/bash
  2
  3 ROOT_UID=0   # Solo gli utenti con $UID 0 posseggono i privilegi di
root.
  4 E_NONROOT=65
  5 E_NOPARAM=66
  6
  7 if [ "$UID" -ne "$ROOT_UID" ]
  8 then
  9   echo "Bisogna essere root per eseguire questo script."
 10   # "Cammina ragazzo, hai finito di poltrire."
 11   exit $E_NONROOT
 12 fi
 13
 14 if [ -z "$1" ]
 15 then
 16   echo "Utilizzo: `basename $0` nome-cercato"
 17   exit $E_NOPARAM
 18 fi
 19
 20
 21 echo "Aggiornamento del database 'locate' ..."
 22 echo "Questo richiede un po' di tempo."
 23 updatedb /usr &     # Deve essere eseguito come root.
 24
 25 wait
 26 #  Non viene eseguita la parte restante dello script finché
'updatedb' non
 27 #+ ha terminato il proprio compito.
 28 #  Si vuole che il database sia aggiornato prima di cercare un nome
di file.
 29
 30 locate $1
 31
```

```
 32 #  Senza il comando wait, nell'ipotesi peggiore, lo script sarebbe
uscito
 33 #+ mentre 'updatedb' era ancora in esecuzione, trasformandolo in un
processo
 34 #+ orfano.
 35
 36 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.

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.

```
  1 #!/bin/bash
  2 # test.sh
  3
  4 ls -l &
  5 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.

```
  1 #!/bin/bash
  2 # test.sh
  3
  4 ls -l &
  5 echo "Fatto."
  6 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 Esempio 13-5).

Esempio 11-24. Uno script che uccide se stesso

```
  1 #!/bin/bash
  2 # self-destruct.sh
  3
  4 kill $$  # Lo script in questo punto "uccide" il suo stesso
processo.
  5          # Ricordo che "$$" è il PID dello script.
  6
  7 echo "Questa riga non viene visualizzata."
  8 # Invece, la shell invia il messaggio "Terminated" allo stdout.
  9
 10 exit 0
 11
 12 #  Dopo che lo script è terminato prematuramente, qual'è l'exit
 13 #+ status restituito?
 14 #
 15 # sh self-destruct.sh
 16 # echo $?
 17 # 143
 18 #
 19 # 143 = 128 + 15
 20 #             segnale SIGTERM
```

`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".

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. [4].

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. [5] Questo fa risparmiare risorse di sistema.

È da notare che **autoload** non fa parte dell'installazione normale di Bash. Bisogna caricarlo con **enable -f** (vedi sopra).

Tabella 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
`$!`	Ultimo processo in background

Note

[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] La libreria *readline* è quella che Bash utilizza per leggere l'input in una shell interattiva.

[4] 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.

[5] Lo stesso risultato di **autoload** può essere ottenuto con typeset -fu.

Capitolo 12. Filtri, programmi e comandi esterni

Sommario

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 principiante 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 utili opzioni sono: `-S`, per ordinare l'elenco in base alla dimensione, `-t`, per ordinarlo in base alla data di modifica e `-i` per mostrare gli inode dei file (vedi Esempio 12-4).

Esempio 12-1. Utilizzare ls per creare un sommario da salvare in un CDR

```
  1 #!/bin/bash
  2 # burn-cd.sh
  3 # Script per rendere automatica la registrazione di un CDR.
  4
  5 VELOC=2           #  Potete utilizzare una velocità più elevata
  6                   #+ se l'hardware la supporta.
  7 FILEIMMAGINE=cdimage.iso
  8 CONTENUTIFILE=contenuti
  9 DISPOSITIVO=cdrom
 10 # DISPOSITIVO="0,0" per le vecchie versioni di cdrecord
 11 DEFAULTDIR=/opt   # Questa è la directory contenente i dati da
registrare.
 12                   # Accertatevi che esista.
 13                   # Esercizio: Aggiungente un controllo che lo
verifichi.
 14
 15 # Viene usato il pacchetto  "cdrecord" di Joerg Schilling:
 16 # http://www.fokus.fhg.de/usr/schilling/cdrecord.html
 17
 18 #  Se questo script viene eseguito da un utente ordinario va
```

```
impostato
 19 #+ il bit suid di cdrecord (chmod u+s /usr/bin/cdrecord, da root).
 20 #  Naturalmente questo crea una falla nella sicurezza, anche se non
rilevante.
 21
 22 if [ -z "$1" ]
 23 then
 24   DIRECTORY_IMMAGINE=$DEFAULTDIR
 25   #  Viene usata la directory predefinita se non ne viene
specificata
 26   #+ alcuna da riga di comando.
 27 else
 28   DIRECTORY_IMMAGINE=$1
 29 fi
 30
 31 #  Crea il "sommario" dei file.
 32 ls -lRF $DIRECTORY_IMMAGINE > $DIRECTORY_IMMAGINE/$CONTENUTIFILE
 33 #  L'opzione "l" fornisce un elenco "dettagliato".
 34 #  L'opzione "R" rende l'elencazione ricorsiva.
 35 #  L'opzione "F" evidenzia i tipi di file (le directory hanno una
 36 #+ "/" dopo il nome).
 37 echo "Il sommario è stato creato."
 38
 39 # Crea l'immagine del file che verrà registrato sul CDR.
 40 mkisofs -r -o $FILEIMMAGINE $DIRECTORY_IMMAGINE
 41 echo "È stata creata l'immagine ($FILEIMMAGINE) su file system
ISO9660."
 42
 43 # Registra il CDR.
 44 cdrecord -v -isosize speed=$VELOC dev=$DISPOSITIVO $FILEIMMAGINE
 45 echo "Sto \"bruciando\" il CD."
 46 echo "Siate pazienti, occorre un po' di tempo."
 47
 48 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.

```
  1 # Usi di 'cat'
  2 cat nomefile                          # Visualizza il contenudo del
file.
  3
  4 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 Esempio 12-24 e Esempio 12-20.


In una pipe, risulta più efficiente redirigere lo `stdin` in un file piuttosto che usare **cat**.

```
  1 cat nomefile | tr a-z A-Z
  2
  3 tr a-z A-Z < nomefile   #  Stesso risultato, ma si avvia
```

```
un processo in meno,
  4                          #+ e senza dover usare la pipe.
```

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 Esempio 12-6).


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 Esempio 9-17 e Esempio A-3.


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.


Il semplice comando **rm** non riesce a cancellare i file i cui nomi iniziano con un trattino.

```
bash$ rm -bruttonome
rm: invalid option -- b
 Try `rm --help' for more information.
```

Il modo per riuscirci è far precedere il nome del file che deve essere rimosso da *punto-barra*.

```
bash$ rm ./-bruttonome
```


Se usato con l'opzione di ricorsività `-r`, il comando cancella tutti i file della directory corrente.

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 Esempio 11-11).

```
  1 chmod +x nomefile
  2 # Rende eseguibile "nomefile" per tutti gli utenti.
  3
  4 chmod u+s nomefile
  5 #  Imposta il bit "suid" di "nomefile".
  6 #  Un utente comune può eseguire "nomefile" con gli stessi privilegi
del
  7 #+ proprietario del file (Non è applicabile agli script di shell).

  1 chmod 644 nomefile
  2 #  Dà al proprietario i permessi di lettura/scrittura su "nomefile",
il
  3 #+ permesso di sola lettura a tutti gli altri utenti
  4 #  (modalità ottale).

  1 chmod 1777 nome-directory
  2 #  Dà a tutti i permessi di lettura, scrittura ed esecuzione nella
  3 #+ directory, inoltre imposta lo "sticky bit". Questo significa che
solo il
  4 #+ proprietario della directory, il proprietario del file e,
naturalmente, root
  5 #+ 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 Esempio 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`.

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 la spiegazione di 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 hard link oppure un link simbolico. 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 reali, 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).

Con i link si ha la possibilità di invocare uno stesso script (o qualsiasi altro eseguibile) con nomi differenti ottenendo un comportamento diverso in base al nome con cui è stato invocato.

Esempio 12-2. Ciao o arrivederci

```
  1 #!/bin/bash
  2 # hello.sh: Visualizzare "ciao" o "arrivederci"
  3 #+          secondo le modalità di invocazione dello script.
  4
  5 # Eseguiamo un collegamento allo script nella directory di lavoro
corrente($PWD):
  6 #    ln -s hello.sh goodbye
  7 # Ora proviamo ad invocare lo script in entrambi i modi:
  8 # ./hello.sh
  9 # ./goodbye
 10
 11
```

```
12 CHIAMATA_CIAO=65
13 CHIAMATA_ARRIVEDERCI=66
14
15 if [ $0 = "./goodbye" ]
16 then
17   echo "Arrivederci!"
18   # Se si desidera, qualche altro saluto dello stesso tipo.
19   exit $CHIAMATA_ARRIVEDERCI
20 fi
21
22 echo "Ciao!"
23 # Qualche altro comando appropriato.
24 exit $CHIAMATA_CIAO
```

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*.

Note

[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.

12.2. Comandi complessi

Comandi per utenti avanzati

find

-exec *COMANDO* \;

Esegue *COMANDO* su ogni file verificato da **find**. La sintassi del comando termina con \; (il ";" deve essere preceduto dal carattere di escape per essere certi che la shell lo passi a **find** col suo significato letterale).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

Se *COMANDO* contiene {}, allora **find** sostituisce "{}" con il percorso completo del file selezionato.

```
  1 find ~/ -name 'core*' -exec rm {} \;
  2 # Cancella tutti i file core presenti nella directory home
dell'utente.
  1 find /home/bozo/projects -mtime 1
```

```
  2 #  Elenca tutti i file della directory /home/bozo/projects
  3 #+ che sono stati modificati il giorno precedente.
  4 #
  5 #  mtime = ora dell'ultima modifica del file in questione
  6 #  ctime = ora dell'ultima modifica di stato (tramite 'chmod' o
altro)
  7 #  atime = ora dell'ultimo accesso
  8
  9 DIR=/home/bozo/junk_files
 10 find "$DIR" -type f -atime +5 -exec rm {} \;
 11 #                                      ^^
 12 #  Le parentesi graffe rappresentano il percorso completo prodotto
da "find."
 13 #
 14 #  Cancella tutti il file in "/home/bozo/junk_files"
 15 #+ a cui non si è acceduto da almeno 5 giorni.
 16 #
 17 #  "-type tipofile", dove
 18 #  f = file regolare
 19 #  d = directory, ecc.
 20 #  (La pagina di manuale di 'find' contiene l'elenco completo.)
```

```
  1 find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-
9][0-9]*' {} \;
  2
  3 #  Trova tutti gli indirizzi IP (xxx.xxx.xxx.xxx) nei file della
directory /etc.
  4 #  Ci sono alcuni caratteri non essenziali - come possono essere
tolti?
  5
  6 # Ecco una possibilità:
  7
  8 find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
  9  | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
 10
 11 # [:digit:] è una delle classi di caratteri
 12 # introdotta con lo standard POSIX 1003.2.
 13
 14 # Grazie, S.C.
```

L'opzione `-exec` di **find** non deve essere confusa con il builtin di shell exec.

Esempio 12-3. Badname, elimina, nella directory corrente, i file i cui nomi contengono caratteri inappropriati e spazi.

```
  1 #!/bin/bash
  2 # badname.sh
  3
  4 # Cancella i file nella directory corrente contenenti caratteri
inadatti.
  5
  6 for nomefile in *
  7 do
  8  nomestrano=`echo "$nomefile" | sed -n
/[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
  9 #nomestrano=`echo "$nomefile" | sed -n '/[+{;"\=?~()<>&*|$]/p'`
anche così.
 10 # Cancella i file contenenti questi caratteri:   + { ; " \ = ? ~ ( )
< > & * | $
 11 #
 12  rm $nomestrano 2>/dev/null
```

```
 13 #              ^^^^^^^^^^  Vengono eliminati i messaggi d'errore.
 14 done
 15
 16 # Ora ci occupiamo dei file contenenti ogni tipo di spaziatura.
 17 find . -name "* *" -exec rm -f {} \;
 18 #  Il percorso del file che "find" cerca prende il posto di "{}".
 19 #  La '\' assicura che il ';' sia interpretato correttamente come
fine del
 20 #+ comando.
 21
 22 exit 0
 23
 24 #-------------------------------------------------------------------
-----
 25 # I seguenti comandi non vengono eseguiti a causa dell'"exit"
precedente.
 26
 27 # Un'alternativa allo script visto prima:
 28 find . -name '*[+{;"\\=?~()<>&*|$ ]*' -exec rm -f '{}' \;
 29 # (Grazie, S.C.)
```

Esempio 12-4. Cancellare un file tramite il suo numero di *inode*

```
  1 #!/bin/bash
  2 # idelete.sh: Cancellare un file per mezzo del suo numero di inode.
  3
  4 #  Questo si rivela utile quando il nome del file inizia con un
  5 #+ carattere non permesso, come ? o -.
  6
  7 ARGCONTO=1          #  Allo script deve essere passato come
argomento
  8                     #+ il nome del file.
  9 E_ERR_ARG=70
 10 E_FILE_NON_ESISTE=71
 11 E_CAMBIO_IDEA=72
 12
 13 if [ $# -ne "$ARGCONTO" ]
 14 then
 15   echo "Utilizzo: `basename $0` nomefile"
 16   exit $E_ERR_ARG
 17 fi
 18
 19 if [ ! -e "$1" ]
 20 then
 21   echo "Il file \""$1"\" non esiste."
 22   exit $E_FILE_NON_ESISTE
 23 fi
 24
 25 inum=`ls -i | grep "$1" | awk '{print $1}'`
 26 #  inum = numero di inode (index node) del file
 27 #  Tutti i file posseggono un inode, la registrazione che contiene
 28 #+ informazioni sul suo indirizzo fisico.
 29
 30 echo; echo -n "Sei assolutamente sicuro di voler cancellare
\"$1\"(s/n)?"
 31 # Anche 'rm' con l'opzione '-v' visualizza la stessa domanda.
 32 read risposta
 33 case "$risposta" in
 34 [nN]) echo "Hai cambiato idea, vero?"
 35       exit $E_CAMBIO_IDEA
 36       ;;
```

```
 37 *)    echo "Cancello il file \"$1\".";;
 38 esac
 39
 40 find . -inum $inum -exec rm {} \;
 41 #                           ^^
 42 #        Le parentesi graffe sono il segnaposto
 43 #+       per il testo prodotto da "find."
 44 echo "Il file "\"$1"\" è stato cancellato!"
 45
 46 exit 0
```

Vedi Esempio 12-25, Esempio 3-4 ed Esempio 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.

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.

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.)*

Esempio 12-5. Creare un file di log utilizzando xargs per verificare i log di sistema

```
 1 #!/bin/bash
 2
 3 #  Genera un file di log nella directory corrente
 4 #+ partendo dalla fine del file /var/log/messages.
 5
 6 #  Nota: /var/log/messages deve avere i permessi di lettura
 7 #+ nel caso lo script venga invocato da un utente ordinario.
 8 #         #root chmod 644 /var/log/messages
 9
10 RIGHE=5
11
12 ( date; uname -a ) >>logfile
13 # Data e nome della macchina
14 echo ------------------------------------------------------------
>>logfile
15 tail -$RIGHE /var/log/messages | xargs |  fmt -s >>logfile
16 echo >>logfile
17 echo >>logfile
18
19 exit 0
20
21 #  Nota:
22 #  ----
23 #  Come ha sottolineato Frank Wang,
24 #+ gli apici non verificati (siano essi singoli o doppi) nel file
sorgente
25 #+ potrebbero far fare indigestione ad xargs.
26 #
27 #  Suggerisce, quindi, di sostituire la riga 15 con la seguente:
28 #     tail -$RIGHE /var/log/messages | tr -d "\"'" | xargs | fmt -s
>>logfile
29
30
31
32 # Esercizio:
33 # ---------
34 #  Modificate lo script in modo che registri i cambiamenti avvenuti
35 #+ in /var/log/messages ad intervalli di venti minuti.
36 #  Suggerimento: usate il comando "watch".
```

Come nel caso di **find**, le due parentesi graffe sostituiscono un testo.

Esempio 12-6. Copiare i file della directory corrente in un'altra

```
 1 #!/bin/bash
 2 # copydir.sh
 3
 4 #  Copia (con dettagli) tutti i file della directory corrente ($PWD)
 5 #+ nella directory specificata da riga di comando.
 6
 7 E_NOARG=65
 8
 9 if [ -z "$1" ]   # Esce se non viene fornito nessun argomento.
10 then
11   echo "Utilizzo: `basename $0` directory-in-cui-copiare"
12   exit $E_NOARG
13 fi
14
```

```
 15 ls . | xargs -i -t cp ./{} $1
 16 #           ^^ ^^      ^^
 17 #  -t è l'opzione "verbose" (invia la riga di comando allo stderr).
 18 #  -i è l'opzione "sostituisci stringhe".
 19 #  {} è il segnaposto del testo di output.
 20 #  E' simile all'uso di una coppia di parentesi graffe in "find."
 21 #
 22 #  Elenca i file presenti nella directory corrente (ls .),
 23 #+ passa l'output di "ls" come argomenti a "xargs" (opzioni -i -t),
 24 #+ quindi copia (cp) questi argomenti ({}) nella nuova directory
($1).
 25
 26 #  Il risultato finale è l'equivalente esatto di
 27 #    cp * $1
 28 # a meno che qualche nome di file contenga caratteri di
"spaziatura".
 29
 30 exit 0
```

Esempio 12-7. Terminare un processo tramite il suo nome

```
  1 #!/bin/bash
  2 # kill-byname.sh: Terminare i processi tramite i loro nomi.
  3 # Confrontate questo script con kill-process.sh.
  4
  5 #  Ad esempio,
  6 #+ provate "./kill-byname.sh xterm" --
  7 #+ e vedrete scomparire dal vostro desktop tutti gli xterm.
  8
  9 #  Attenzione:
 10 #  ----------
 11 #  Si tratta di uno script veramente pericoloso.
 12 #  Eseguirlo distrattamente (specialmente da root)
 13 #+ può causare perdita di dati ed altri effetti indesiderati.
 14
 15 E_NOARG=66
 16
 17 if test -z "$1"  # Nessun argomento fornito da riga di comando?
 18 then
 19   echo "Utilizzo: `basename $0` Processo(i)_da_terminare"
 20   exit $E_NOARG
 21 fi
 22
 23
 24 NOME_PROCESSO="$1"
 25 ps ax | grep "$NOME_PROCESSO" | awk '{print $1}' | xargs -i kill {}
2&>/dev/null
 26 #                                                    ^^      ^^
 27
 28 # ------------------------------------------------------------------
--
 29 # Note:
 30 # -i è l'opzione "sostituisci stringhe" di xargs.
 31 # Le parentesi graffe rappresentano il segnaposto per la
sostituzione.
 32 # 2&>/dev/null elimina i messaggi d'errore indesiderati.
 33 # ------------------------------------------------------------------
--
 34
 35 exit $?
```

Esempio 12-8. Analisi di frequenza delle parole utilizzando xargs

```
  1 #!/bin/bash
  2 # wf2.sh: Analisi sommaria della frequenza delle parole in un file
di testo.
  3
  4 # Usa 'xargs' per scomporre le righe del testo in parole singole.
  5 # Confrontate quest'esempio con lo script "wf.sh" che viene dopo.
  6
  7
  8 # Verifica la presenza di un file di input passato da riga di
comando.
  9 ARG=1
 10 E_ERR_ARG=65
 11 E_NOFILE=66
 12
 13 if [ $# -ne "$ARG" ]
 14 # Il numero di argomenti passati allo script è corretto?
 15 then
 16   echo "Utilizzo: `basename $0` nomefile"
 17   exit $E_ERR_ARG
 18 fi
 19
 20 if [ ! -f "$1" ]       # Verifica se il file esiste.
 21 then
 22   echo "Il file \"$1\" non esiste."
 23   exit $E_NOFILE
 24 fi
 25
 26
 27
 28 ########################################################
 29 cat "$1" | xargs -n1 | \
 30 #  Elenca il file una parola per riga.
 31 tr A-Z a-z | \
 32 #  Cambia tutte le lettere maiuscole in minuscole.
 33 sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
 34 /g' | \
 35 #  Filtra i punti e le virgole, e
 36 #+ cambia gli spazi tra le parole in linefeed.
 37 sort | uniq -c | sort -nr
 38 #  Infine premette il conteggio delle occorrenze e le
 39 #+ ordina in base al numero.
 40 ########################################################
 41
 42 #  Svolge lo stesso lavoro dell'esempio "wf.sh",
 43 #+ ma in modo un po' più greve e lento.
 44
 45 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, logiche, su stringhe o confronti.

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.

Esempio 12-9. Utilizzo di expr

```
 1 #!/bin/bash
 2
 3 # Dimostrazione di alcuni degli usi di 'expr'
 4 # ============================================
 5
 6 echo
 7
 8 # Operatori aritmetici
 9 # --------- ----------
10
11 echo "Operatori aritmetici"
12 echo
13 a=`expr 5 + 3`
14 echo "5 + 3 = $a"
15
16 a=`expr $a + 1`
17 echo
18 echo "a + 1 = $a"
19 echo "(incremento di variabile)"
20
21 a=`expr 5 % 3`
22 # modulo
23 echo
24 echo "5 modulo 3 = $a"
25
26 echo
27 echo
28
29 # Operatori logici
30 # --------- ------
31
32 #  Restituisce 1 per vero, 0 per falso,
33 #+ il contrario della normale convenzione Bash.
34
35 echo "Operatori logici"
36 echo
37
38 x=24
```

```
 39 y=25
 40 b=`expr $x = $y`         # Verifica l'uguaglianza.
 41 echo "b = $b"            # 0  ( $x -ne $y )
 42 echo
 43
 44 a=3
 45 b=`expr $a \> 10`
 46 echo 'b=`expr $a \> 10`, quindi...'
 47 echo "Se a > 10, b = 0 ((falso)"
 48 echo "b = $b"            # 0  ( 3 ! -gt 10 )
 49 echo
 50
 51 b=`expr $a \< 10`
 52 echo "Se a < 10, b = 1 (vero)"
 53 echo "b = $b"            # 1  ( 3 -lt 10 )
 54 echo
 55 # Notate l'uso dell'escaping degli operatori.
 56
 57 b=`expr $a \<= 3`
 58 echo "Se a <= 3, b = 1 (vero)"
 59 echo "b = $b"            # 1  ( 3 -le 3 )
 60 # Esiste anche l'operatore "\>=" (maggiore di o uguale a).
 61
 62
 63 echo
 64 echo
 65
 66
 67
 68 # Operatori per stringhe
 69 # --------- --- --------
 70
 71 echo "Operatori per stringhe"
 72 echo
 73
 74 a=1234zipper43231
 75 echo "La stringa su cui opereremo è \"$a\"."
 76
 77 # length: lunghezza della stringa
 78 b=`expr length $a`
 79 echo "La lunghezza di \"$a\" è $b."
 80
 81 # index: posizione, in stringa, del primo carattere
 82 #        della sottostringa verificato
 83 b=`expr index $a 23`
 84 echo "La posizione numerica del primo \"2\" in \"$a\" è \"$b\"."
 85
 86 #  substr: estrae una sottostringa, iniziando da posizione &
lunghezza
 87 #+ specificate
 88 b=`expr substr $a 2 6`
 89 echo "La sottostringa di \"$a\", iniziando dalla posizione 2,\
 90 e con lunghezza 6 caratteri è \"$b\"."
 91
 92
 93 #  Il comportamento preimpostato delle operazioni 'match' è quello
 94 #+ di cercare l'occorrenza specificata all'***inizio*** della
stringa.
 95 #
 96 #        usa le Espressioni Regolari
 97 b=`expr match "$a" '[0-9]*'`     #  Conteggio numerico.
 98 echo "Il numero di cifre all'inizio di \"$a\" è $b."
```

```
 99 b=`expr match "$a" '\([0-9]*\)'` #  Notate che le parentesi con
l'escape
100 #                  ==      ==   #+ consentono la verifica della
sottostringa.
101 echo "Le cifre all'inizio di \"$a\" sono \"$b\"."
102
103 echo
104
105 exit 0
```

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.

```
  1 #!/bin/bash
  2
  3 echo
  4 echo "Operazioni sulle stringhe usando il costrutto
\"expr \$stringa : \""
  5 echo
"================================================="
  6 echo
  7
  8 a=1234zipper5FLIPPER43231
  9
 10 echo "La stringa su cui opereremo è \"`expr "$a" :
'\(.*\)'`\"."
 11 # Operatore di raggruppamento parentesi con escape. ==
==
 12
 13 #       ************************
 14 #+       Le parentesi con l'escape
 15 #+      verificano una sottostringa
 16 #       ************************
 17
 18
 19 #  Se non si esegue l'escaping delle parentesi...
 20 #+ allora 'expr' converte l'operando stringa in un
intero.
 21
 22 echo "La lunghezza di \"$a\" è `expr "$a" : '.*'`."#
Lunghezza della stringa
 23
 24 echo "Il numero di cifre all'inizio di \"$a\" è `expr
"$a" : '[0-9]*'`."
 25
 26 # ------------------------------------------------------
----------------- #
 27
 28 echo
 29
 30 echo "Le cifre all'inizio di \"$a\" sono `expr "$a" :
'\([0-9]*\)'`."
 31 #                                                   ==
==
 32 echo "I primi 7 caratteri di \"$a\" sono `expr "$a" :
'\(.......\)'`."
 33 #       =====                                       ==
==
 34 # Ancora, le parentesi con l'escape forzano la verifica
della sottostringa.
 35 #
 36 echo"Gli ultimi 7 caratteri di \"$a\" sono `expr "$a" :
```

```
'.*\(.......\)'`."
 37 #        ======              operatore di fine stringa
^^
 38 #  (in realtà questo vuol dire saltare uno o più
caratteri finché non viene
 39 #+ raggiunta la sottostringa specificata)
 40
 41 echo
 42
 43 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 la Sezione 34.2) è un'alternativa attraente all'uso di **expr**.

Vedi la Sezione 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.

Esempio 12-10. Utilizzo di date

```
  1 #!/bin/bash
  2 # Esercitarsi con il comando 'date'
  3
  4 echo "Il numero di giorni trascorsi dall'inizio dell'anno è `date
+%j`."
  5 # È necessario il '+' per il formato dell'output.
  6 # %j fornisce i giorni dall'inizio dell'anno.
  7
  8 echo "Il numero di secondi trascorsi dal 01/01/1970 è `date +%s`."
  9 #  %s contiene il numero di secondi dall'inizio della "UNIX epoch",
ma
 10 #+ quanto può essere utile?
 11
 12 prefisso=temp
 13 suffisso=$(date +%s)  # L'opzione "+%s" di 'date' è una specifica
GNU.
 14 nomefile=$prefisso.$suffisso
 15 echo $nomefile
 16 #  È importantissima per creare nomi di file temporanei "unici", è
persino
 17 #+ migliore dell'uso di $$.
 18
 19 # Leggete la pagina di manuale di 'date' per le altre opzioni di
formato.
```

```
 20
 21 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.

```
  1 date +%N | sed -e 's/000$//' -e 's/^0//'
  2            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  3 # Toglie gli zeri iniziali e finali, se presenti.
```

Esistono molte altre opzioni (eseguite **man date**).

```
  1 date +%j
  2 # Visualizza i giorni dell'anno (i giorni trascorsi dal 1 gennaio).
  3
  4 date +%k%M
  5 # Visualizza l'ora e i minuti nel formato 24-ore, come unica stringa
di cifre.
```

Vedi anche Esempio 3-4.

zdump

Controllo dell'ora di zona: visualizza l'ora di una zona specificata.

```
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.

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.

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 cron. 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, con l'opzione `-f`, i comandi da un file.

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 Esempio 30-6.

```
1 sleep 3     # Pausa di 3 secondi.
```


Il comando **sleep** conta, in modo predefinito, i secondi. Possono però essere specificati minuti, ore o giorni.

```
1 sleep 3 h    # Pausa di 3 ore!
```


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.

```
1 usleep 30     # Pausa di 30 microsecondi.
```

Questo comando fa parte del pacchetto Red Hat *initscripts / rc-scripts*.


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 per l'elaborazione del 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* ne elenca le funzionalità e le molteplici opzioni. Vedi Esempio 10-9, Esempio 10-10 e Esempio A-9.

tsort

Esegue un ordinamento topologico di stringhe lette in coppia secondo i modelli forniti nell'input.

uniq

Questo filtro elimina le righe duplicate di un file che è stato ordinato. È spesso usato in una pipe in coppia con sort.

```
  1 cat lista-1 lista-2 lista-3 | sort | uniq > listafinale
  2 # Vengono concatenati i file lista,
  3 # ordinati,
  4 # eliminate le righe doppie,
  5 # ed infine il risultato viene scritto in un file di output.
```

L'opzione `-c` premette ad ogni riga del file di input il numero delle sue occorrenze.

```
bash$ cat fileprova
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 fileprova
      1 Questa riga è presente una sola volta.
      2 Questa riga è presente due volte.
      3 Questa riga è presente tre volte.


bash$ sort fileprova | 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.

Esempio 12-11. Analisi delle frequenze delle parole

```
  1 #!/bin/bash
  2 #  wf.sh: Un'analisi sommaria, su un file di testo, della
  3 #+ frequenza delle parole.
  4 #  È una versione più efficiente dello script "wf2.sh".
  5
  6
  7 # Verifica la presenza di un file di input passato da riga di
comando.
  8 ARG=1
  9 E_ERR_ARG=65
 10 E_NOFILE=66
 11
 12 if [ $# -ne "$ARG" ]  #  Il numero di argomenti passati allo script è
corretto?
 13 then
 14   echo "Utilizzo: `basename $0` nomefile"
 15   exit $E_ERR_ARG
 16 fi
 17
 18 if [ ! -f "$1" ]      # Verifica se il file esiste.
 19 then
```

```
 20   echo "Il file \"$1\" non esiste."
 21   exit $E_NOFILE
 22 fi
 23
 24
 25

26 ###############################################################################
#########
 27 # main ()
 28 sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
 29 /g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
 30 #                           ========================
 31 #                           Frequenza delle occorrenze
 32
 33 #  Filtra i punti e le virgole, e cambia gli spazi tra le parole in
 34 #+ linefeed, quindi trasforma tutti i caratteri in caratteri
minuscoli ed
 35 #+ infine premette il conteggio delle occorrenze e le ordina in base
al numero.
 36
 37 #  Arun Giridhar suggerisce di modificare il precedente in:
 38 #  . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
 39 #  In questo modo viene aggiunta una chiave di ordinamento
secondaria, per cui
 40 #+ nel caso di occorrenze uguali queste vengono ordinate
alfabeticamente.
 41 #  Ecco la spiegazione:
 42 #  "Effettivamente si tratta di un ordinamento di radice, prima sulla
 43 #+ colonna meno significativa
 44 #+ (parola o stringa, opzionalmente senza distinzione minuscolo-
maiuscolo)
 45 #+ infine sulla colonna più significativa (frequenza)."

46 ###############################################################################
#########
 47
 48 exit 0
 49
 50 # Esercizi:
 51 # ---------
 52 # 1)  Aggiungete dei comandi a 'sed' per filtrare altri segni di
 53 #   + punteggiatura, come i punti e virgola.
 54 # 2)  Modificatelo per filtrare anche gli spazi multipli e gli altri
 55 #   + caratteri di spaziatura.
```

```
bash$ cat fileprova
 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 fileprova
       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:

```
  1 cat /etc/mtab | cut -d ' ' -f1,2
```

Usare **cut** per visualizzare la versione del SO e del kernel:

```
  1 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:

```
  1 # Elenca tutti gli utenti presenti nel file /etc/passwd.
  2
  3 FILE=/etc/passwd
  4
  5 for utente in $(cut -d: -f1 $FILE)
  6 do
  7   echo $utente
  8 done
  9
 10 # Grazie, Oleg Philon per il suggerimento.
```

`cut -d ' ' -f2,3 nomefile` equivale a **`awk -F'[ ]' '{ print $2, $3 }' nomefile`**

Vedi anche Esempio 12-39.

paste

Strumento per unire più file in un unico file impaginato su 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.

```
  1 File: 1.dat
  2
  3 100 Scarpe
  4 200 Lacci
  5 300 Calze
```

```
  1 File: 2.dat
  2
  3 100 EUR 40.00
  4 200 EUR 1.00
  5 300 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
```


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.

Esempio 12-12. Quali file sono degli script?

```
  1 #!/bin/bash
  2 # script-detector.sh: Rileva gli script presenti in una directory.
  3
  4 VERCAR=2          # Verifica i primi 2 caratteri.
  5 INTERPRETE='#!'   # Gli script iniziano con "#!".
  6
  7 for file in *     # Verifica tutti i file della directory corrente.
  8 do
  9   if [[ `head -c$VERCAR "$file"` = "$INTERPRETE" ]]
 10   #      head -c2                   #!
 11   #  L'opzione '-c' di "head" agisce sul numero di caratteri
specificato
 12   #+ anzichè sulle righe (comportamento di default).
 13   then
 14       echo "Il file \"$file\" è uno script."
 15   else
```

```
 16       echo "Il file \"$file\"  *non* è uno script."
 17   fi
 18 done
 19
 20 exit 0
```

Esempio 12-13. Generare numeri casuali di 10 cifre

```
  1 #!/bin/bash
  2 # rnd.sh: Visualizza un numero casuale di 10 cifre
  3
  4 # Script di Stephane Chazelas.
  5
  6 head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
  7
  8
  9 #
================================================================== #
 10
 11 # Analisi
 12 # --------
 13
 14 # head:
 15 # l'opzione -c4 considera solamente i primi 4 byte.
 16
 17 # od:
 18 # L'opzione -N4 limita l'output a 4 byte.
 19 # L'opzione -tu4 seleziona, per l'output, il formato decimale senza
segno.
 20
 21 # sed:
 22 #  L'opzione -n, in combinazione con l'opzione "p" del comando "s",
prende
 23 #+ in considerazione, per l'output, solo le righe verificate.
 24
 25
 26
 27 # L'autore di questo script spiega l'azione di 'sed' come segue.
 28
 29 # head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
 30 # ---------------------------------> |
 31
 32 # Assumiamo che l'output fino a "sed">|
 33 # sia 0000000 1198195154\n
 34
 35 #  sed inizia leggendo i caratteri: 0000000 1198195154\n.
 36 #  Qui trova il carattere di ritorno a capo, quindi è pronto per
elaborare
 37 #+ la prima riga (0000000 1198195154), che assomiglia alla sua
direttiva
 38 #+ <righe><comandi>. La prima ed unica è
 39
 40 #   righe     comandi
 41 #   1         s/.* //p
 42
 43 #  Il numero della riga è nell'intervallo, quindi entra in azione:
 44 #+ cerca di sostituire la stringa più lunga terminante con uno
spazio
 45 #+ ("0000000 ") con niente "//" e in caso di successo, visualizza il
risultato
 46 #+ ("p" è l'opzione del comando "s", ed è differente dal comando
```

```
"p").
 47
 48 #  sed ora continua la lettura dell'input. (Notate che prima di
continuare, se
 49 #+ non fosse stata passata l'opzione -n, sed avrebbe visualizzato la
riga
 50 #+ un'altra volta).
 51
 52 #  sed adesso legge la parte di caratteri rimanente, e trova la fine
del file.
 53 #  Si appresta ad elaborare la seconda riga (che può anche essere
numerata
 54 #+ con '$' perché è l'ultima).
 55 #  Constata che non è compresa in <righe> e quindi termina il
lavoro.
 56
 57 #  In poche parole, questo comando sed significa: "Solo sulla prima
riga, togli
 58 #+ qualsiasi carattere fino allo spazio, quindi visualizza il
resto."
 59
 60 # Un modo migliore per ottenere lo stesso risultato sarebbe stato:
 61 #           sed -e 's/.* //;q'
 62
 63 #  Qui abbiamo due <righe> e due <comandi> (si sarebbe potuto anche
scrivere
 64 #           sed -e 's/.* //' -e q):
 65
 66 #   righe                       comandi
 67 #   niente (verifica la riga)   s/.* //
 68 #   niente (verifica la riga)   q (quit)
 69
 70 #  In questo esempio, sed legge solo la sua prima riga di input.
 71 #  Esegue entrambi i comandi e visualizza la riga (con la
sostituzione) prima
 72 #+ di uscire (a causa del comando "q"), perché non gli è stata
passata
 73 #+ l'opzione "-n".
 74
 75 #
================================================================== #
 76
 77 #  Un'alternativa più semplice al precedente script,
 78 #+ formata da una sola riga, potrebbe essere:
 79 #          head -c4 /dev/urandom| od -An -tu4
 80
 81 exit 0
```

Vedi anche Esempio 12-33.

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 di visualizzare le righe accodate al file.

Esempio 12-14. Utilizzare tail per controllare il log di sistema

```
  1 #!/bin/bash
  2
  3 nomefile=sys.log
  4
```

```
  5 cat /dev/null > $nomefile; echo "Creazione / cancellazione del
file."
  6 #  Crea il file nel caso non esista, mentre lo svuota se è già stato
creato.
  7 #  vanno bene anche : > nomefile   e   > nomefile.
  8
  9 tail /var/log/messages > $nomefile
 10 # /var/log/messages deve avere i permessi di lettura perché lo
script funzioni.
 11
 12 echo "$nomefile contiene la parte finale del log di sistema."
 13
 14 exit 0
```

Vedi anche Esempio 12-5, Esempio 12-33 e Esempio 30-6.

grep

Strumento di ricerca multifunzione che fa uso delle Espressioni Regolari. In origine era un comando/filtro del venerabile editor di linea **ed**: **`g/re/p`** -- *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 esatte.

L'opzione `-l` elenca solo i file in cui la ricerca ha avuto successo, ma non le righe verificate.

L'opzione `-r` (ricorsivo) 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.

```
  1 grep modello1 *.txt | grep -v modello2
  2
  3 # Verifica tutte le righe dei file "*.txt" contenenti "modello1",
  4 # ma ***non*** quelle contenenti "modello2".
```

L'opzione `-c` (`--count`) fornisce il numero delle occorrenze, ma non le visualizza.

```
  1 grep -c txt *.sgml   # ((numero di occorrenze di "txt" nei file
"*.sgml")
  2
  3
  4 #   grep -cz .
  5 #            ^ punto
  6 # significa conteggio (-c) zero-diviso (-z) elementi da cercare "."
  7 # cioè, quelli non vuoti (contenenti almeno 1 carattere).
  8 #
  9 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .
# 4
 10 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'
# 5
 11 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'
# 5
 12 #
 13 printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'
# 9
 14 # Per default, i caratteri di a capo (\n) separano gli elementi da
cercare.
 15
 16 # Notate che l'opzione -z è specifica del "grep" di GNU.
 17
 18
 19 # 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.
```

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.

```
 1 SUCCESSO=0                      # se la ricerca di grep è riuscita
 2 parola=Linux
 3 file=file.dati
 4
 5 grep -q "$parola" "$file"
 6 # L'opzione "-q" non visualizza nulla allo stdout.
 7
 8 if [ $? -eq $SUCCESSO ]
 9 # if grep -q "$parola" "$file"   può sostituire le righe 5 - 8.
10 then
11   echo "$parola è presente in $file"
12 else
13   echo "$parola non è presente in $file"
14 fi
```

L'Esempio 30-6 dimostra come usare **grep** per cercare una parola in un file di log di sistema.

Esempio 12-15. Simulare "grep" in uno script

```
 1 #!/bin/bash
 2 # grp.sh: Una reimplementazione molto sommaria di 'grep'.
 3
 4 E_ERR_ARG=65
 5
 6 if [ -z "$1" ]    # Verifica se sono stati passati argomenti allo
script.
 7 then
 8   echo "Utilizzo: `basename $0` modello"
 9   exit $E_ERR_ARG
10 fi
11
12 echo
13
14 for file in *     # Verifica tutti i file in $PWD.
15 do
16   output=$(sed -n /"$1"/p $file)  # Sostituzione di comando.
17
18   if [ ! -z "$output" ]           #  Cosa succede se si usa
"$output"
19                                   #+ senza i doppi apici?
20   then
21       echo -n "$file: "
22       echo $output
23   fi              # sed -ne "/$1/s|^|${file}: |p" equivale al
precedente.
24
25   echo
26 done
27
28 echo
29
30 exit 0
31
32 # Esercizi:
33 # ---------
34 #  1) Aggiungete nuove righe di output nel caso ci sia più di una
```

```
35 #+    occorrenza per il file dato.
36 #  2) Aggiungete altre funzionalità.
```

È possibile far ricercare a **grep** due differenti modelli? Cosa si può fare se volessimo che **grep** visualizzi tutte le righe di un file o i file che contengono sia "modello1" *che* "modello2"?

Un metodo per raggiungere questo scopo è quello di collegare con una pipe il risultato di **grep modello1** a **grep modello2**.

```
 1 # tstfile
 2
 3 Questo è un file d'esempio.
 4 Questo è un file di testo ordinario.
 5 Questo file non contiene testo strano.
 6 Questo file non è insolito.
 7 Altro testo.
```

```
bash$ grep file tstfile
# tstfile
Questo è un file d'esempio.
Questo è un file di testo ordinario.
Questo file non contiene testo strano.
Questo file non è insolito..

bash$ grep file tstfile | grep testo
Questo è un file di testo ordinario.
Questo file non contiene testo strano.
```

egrep (*extended grep*) grep esteso, 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 (*fast grep*) grep veloce, è uguale a **grep -F**. Esegue la ricerca letterale della stringa (niente espressioni regolari), il che accelera sensibilmente l'operazione.

agrep (*approximate grep*) grep d'approssimazione, 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.

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.

Esempio 12-16. Verificare la validità delle parole con un dizionario

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

```
  2 # lookup: Esegue una verifica di dizionario di tutte le parole di un
file dati.
  3
  4 file=file.dati     # File dati le cui parole devono essere
controllate.
  5
  6 echo
  7
  8 while [ "$Parola" != fine ] # Ultima parola del file dati.
  9 do
 10   read parola      #  Dal file dati, a seguito della redirezione a
fine ciclo.
 11   look $parola > /dev/null  #  Per non visualizzare le righe del
 12                             #+ file dizionario.
 13   verifica=$?      # Exit status del comando 'look'.
 14
 15   if [ "$verifica" -eq 0 ]
 16   then
 17      echo "\"$parola\" è valida."
 18   else
 19      echo "\"$parola\" non è valida."
 20   fi
 21
 22 done <"$file"      #  Redirige lo stdin a $file, in modo che "read"
agisca
 23                    #+ su questo.
 24
 25 echo
 26
 27 exit 0
 28
 29 # ------------------------------------------------------------------
 30 #  Le righe di codice seguenti non vengono eseguite a causa del
 31 #+ precedente comando "exit".
 32
 33
 34 # Stephane Chazelas propone la seguente, e più concisa, alternativa:
 35
 36 while read parola && [[ $parola != fine ]]
 37 do if look "$parola" > /dev/null
 38    then echo "\"$parola\" è valida."
 39    else echo "\"$parola\" non è valida."
 40    fi
 41 done <"$file"
 42
 43 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:

```
  1 $ ls *.txt | wc -l
  2 #  Il conteggio si interrompe se viene trovato un carattere di
  3 #+ linefeed nel nome di uno dei file "*.txt".
  4
  5 # Modi alternativi per svolgere lo stesso compito:
  6 #      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
  7 #      (shopt -s nullglob; set -- *.txt; echo $#)
  8
  9 # 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 Esempio 12-33 e Esempio 16-8.

Alcuni comandi possiedono, sotto forma di opzioni, alcune delle funzionalità di **wc**.

```
  1 ... | grep foo | wc -l
  2 #  Questo costrutto, frequentemente usato, può essere reso in modo
più conciso.
  3
  4 ... | grep -c foo
```

```
  5 # Un semplice impiego dell'opzione "-c" (o "--count") di grep.
  6
  7 # Grazie, S.C.
```

tr

filtro per la sostituzione di caratteri.

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.

```
  1 echo "abcdef"                 # abcdef
  2 echo "abcdef" | tr -d b-d     # aef
  3
  4
  5 tr -d 0-9 <nomefile
  6 # 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. [1]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Esempio 12-17. toupper: Trasforma tutte le lettere di un file in maiuscole

```
  1 #!/bin/bash
  2 # Modifica tutte le lettere del file in maiuscole.
  3
  4 E_ERR_ARG=65
  5
  6 if [ -z "$1" ]  # Verifica standard degli argomenti da riga di
comando.
  7 then
```

```
  8   echo "Utilizzo: `basename $0` nomefile"
  9   exit $E_ERR_ARG
 10 fi
 11
 12 tr a-z A-Z <"$1"
 13
 14 # Stesso effetto del precedente, ma usando la notazione POSIX:
 15 #        tr '[:lower:]' '[:upper:]' <"$1"
 16 # Grazie, S.C.
 17
 18
 19 exit 0
```

Esempio 12-18. lowercase: Modifica tutti i nomi dei file della directory corrente in lettere minuscole

```
  1 #! /bin/bash
  2 #
  3 #  Cambia ogni nome di file della directory di lavoro in lettere
minuscole.
  4 #
  5 #  Ispirato da uno script di John Dubois, che è stato tradotto in
Bash da Chet
  6 #+ Ramey e semplificato considerevolmente da Mendel Cooper,
  7 #+ autore del presente libro.
  8
  9
 10 for file in *                   # Controlla tutti i file della
directory.
 11 do
 12    fnome=`basename $file`
 13    n=`echo $fnome | tr A-Z a-z`  #  Cambia il nome del file in tutte
 14                                  #+ lettere minuscole.
 15    if [ "$fnome" != "$n" ]       #  Rinomina solo quei file che non
 16                                  #+ sono già in minuscolo.
 17    then
 18       mv $fnome $n
 19    fi
 20 done
 21
 22 exit 0
 23
 24
 25 #  Il codice che si trova oltre questa riga non viene eseguito a
causa
 26 #+ del precedente "exit".
 27 #-------------------------------------------------------------------
--#
 28 # Se volete eseguirlo, cancellate o commentate le righe precedenti.
 29
 30 #  Lo script visto sopra non funziona con nomi di file conteneti
spazi
 31 #+ o ritorni a capo.
 32
 33 # Stephane Chazelas, quindi, suggerisce l'alternativa seguente:
 34
 35
 36 for file in *        #  Non è necessario usare basename, perché "*"
non
 37                     #+ restituisce i nomi di file contenenti "/".
 38
```

```
 39 do n=`echo "$file/" | tr '[:upper:]' '[:lower:]'`
 40
 41 #                    Notazione POSIX dei set di caratteri.
 42 #                    È stata aggiunta una barra, in modo che gli
 43 #                    eventuali ritorni a capo non vengano cancellati
 44 #                    dalla sostituzione di comando.
 45   # Sostituzione di variabile:
 46   n=${n%/}           #  Rimuove le barre, aggiunte precedentemente,
dal
 47                      #+ nome del file.
 48
 49   [[ $file == $n ]] || mv "$file" "$n"
 50
 51                      # Verifica se il nome del file è già in
minuscolo.
 52
 53 done
 54
 55 exit 0
```

Esempio 12-19. Du: Conversione di file di testo DOS al formato UNIX

```
  1 #!/bin/bash
  2 # Du.sh: converte i file di testo DOS in formato UNIX .
  3
  4 E_ERR_ARG=65
  5
  6 if [ -z "$1" ]
  7 then
  8   echo "Utilizzo: `basename $0` nomefile-da-convertire"
  9   exit $E_ERR_ARG
 10 fi
 11
 12 NUOVONOMEFILE=$1.unx
 13
 14 CR='\015'  # Ritorno a capo.
 15            # 015 è il codice ottale ASCII di CR
 16            # Le righe dei file di testo DOS terminano con un CR-LF.
 17            # Le righe dei file di testo UNIX terminano con il solo
LF.
 18
 19 tr -d $CR < $1 > $NUOVONOMEFILE
 20 # Cancella i CR e scrive il file nuovo.
 21
 22 echo "Il file di testo originale DOS è \"$1\"."
 23 echo "Il file di testo tradotto in formato UNIX è
\"$NOMENUOVOFILE\"."
 24
 25 exit 0
 26
 27 # Esercizio:
 28 #-----------
 29 # Modificate lo script per la conversione inversa (da UNIX a DOS).
```

Esempio 12-20. rot13: cifratura ultra-debole

```
  1 #!/bin/bash
  2 # rot13.sh: Classico algoritmo rot13, cifratura che potrebbe beffare
solo un
  3 #           bambino di 3 anni.
```

```
  4
  5 # Utilizzo: ./rot13.sh nomefile
  6 # o      ./rot13.sh <nomefile
  7 # o      ./rot13.sh e fornire l'input da tastiera (stdin)
  8
  9 cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'  # "a" corrisponde a "n", "b"
a "o", ecc.
 10 #  Il costrutto 'cat "$@"' consente di gestire un input proveniente
sia dallo
 11 #+ stdin che da un file.
 12
 13 exit 0
```

Esempio 12-21. Generare "Rompicapi Cifrati" di frasi celebri

```
  1 #!/bin/bash
  2 # crypto-quote.sh: Cifra citazioni
  3
  4 #  Cifra frasi famose mediante una semplice sostituzione
monoalfabetica.
  5 #  Il risultato è simile ai rompicapo "Crypto Quote" delle pagine
Op Ed
  6 #+ del Sunday.
  7
  8
  9 chiave=ETAOINSHRDLUBCFGJMQPVWZYXK
 10 # La "chiave" non è nient'altro che l'alfabeto rimescolato.
 11 # Modificando la "chiave" cambia la cifratura.
 12
 13 # Il costrutto 'cat "$@"' permette l'input sia dallo stdin che dai
file.
 14 # Se si usa lo stdin, l'input va terminato con un Control-D.
 15 # Altrimenti occorre specificare il nome del file come parametro
da riga
 16 # di comando.
 17
 18 cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$chiave"
 19 #        |  in maiuscolo  |     cifra
 20 # Funziona con frasi formate da lettere minuscole, maiuscole o
entrambe.
 21 # I caratteri non alfabetici non vengono modificati.
 22
 23
 24 # Provate lo script con qualcosa di simile a
 25 # "Nothing so needs reforming as other people's habits."
 26 # --Mark Twain
 27 #
 28 # Il risultato è:
 29 # "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
 30 # --BEML PZERC
 31
 32 # Per decodificarlo:
 33 # cat "$@" | tr "$chiave" "A-Z"
 34
 35
 36 #  Questa semplice cifratura può essere spezzata da un dodicenne
con il
 37 #+ semplice uso di carta e penna.
 38
 39 exit 0
 40
```

```
 41 #  Esercizio:
 42 #  ---------
 43 #  Modificate lo script in modo che sia in grado sia di cifrare
che di
 44 #+ decifrare, in base al(i) argomento(i) passato(i) da riga di
comando.
```

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 Esempio 12-22 e Esempio A-2).

fmt

Semplice formattatore di file usato come filtro, in una pipe, per "ridimensionare" lunghe righe di testo per l'output.

Esempio 12-22. Dimensionare un elenco di file

```
 1 #!/bin/bash
 2
 3 AMPIEZZA=40                # Ampiezza di 40 colonne.
 4
 5 b=`ls /usr/local/bin`      # Esegue l'elenco dei file...
 6
 7 echo $b | fmt -w $AMPIEZZA
 8
 9 # Si sarebbe potuto fare anche con
10 #    echo $b | fold - -s -w $AMPIEZZA
11
12 exit 0
```

Vedi anche Esempio 12-5.

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.

Esempio 12-23. Utilizzo di column per impaginare un elenco di directory

```
  1 #!/bin/bash
  2 #  L'esempio seguente corrisponde, con piccole modifiche, a quello
  3 #+ contenuto nella pagina di manuale di "column".
  4
  5 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-
NAME\n" \
  6 ; ls -l | sed 1d) | column -t
  7
  8 #  "sed 1d" nella pipe cancella la prima riga di output, che sarebbe
  9 #+ "total        N",
 10 #+ dove "N" è il numero totale di file elencati da "ls -l".
 11
 12 # L'opzione -t di "column" visualizza l'output in forma tabellare.
 13
 14 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`.

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.

Esempio 12-24. nl: Uno script che numera le proprie righe

```
  1 #!/bin/bash
  2
  3 # Questo script si auto-visualizza due volte con le righe numerate.
  4
  5 #  'nl' considera questa riga come la nr. 3 perché le righe
  6 #+ vuote vengono saltate.
  7 # 'cat -n' vede la riga precedente come la numero 5.
  8
  9 nl `basename $0`
 10
 11 echo; echo  # Ora proviamo con 'cat -n'
 12
 13 cat -n `basename $0`
 14 # La differenza è che 'cat -n' numera le righe vuote.
```

```
15 # Notate che lo stesso risultato lo si ottiene con  'nl -ba'.
16
17 exit 0
18 #-----------------------------------------------------------------
```

pr

Filtro di formato di visualizzazione. Impagina i file (o lo `stdout`) in sezioni adatte alla visualizzazione su schermo o 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. Spesso risulta conveniente 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 Esempio 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**.

Note

[1] Questo è vero solo per la versione GNU di **tr**, non per la versione generica che si trova spesso sui sistemi commerciali UNIX

12.5. Comandi per i file e l'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 Esempio 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**). [1]

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)


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` comprime l'archivio con bzip2

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 stata sostituita 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 gli altri, 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.

`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.

Esempio 12-25. Utilizzo di cpio per spostare una directory

```
 1 #!/bin/bash
 2
 3 # Copiare una directory usando 'cpio.'
 4
 5 ARG=2
 6 E_ERR_ARG=65
 7
 8 if [ $# -ne "$ARG" ]
 9 then
10   echo "Utilizzo: `basename $0` directory_origine
directory_destinazione"
11   exit $E_ERR_ARG
12 fi
13
14 origine=$1
15 destinazione=$2
16
17 find "$origine" -depth | cpio -admvp "$destinazione"
18 #                ^^^^^         ^^^^^
19 # Leggete le pagine di manuale di 'find' e 'cpio' per decifrare
queste opzioni.
20
21 #  Qui si potrebbe inserire il controllo dell'exit status ($?)
22 #+ per vedere se tutto ha funzionato correttamente.
23
24 exit 0
```

rpm2cpio

Questo comando crea un archivio **cpio** da un archivio rpm.

Esempio 12-26. Decomprimere un archivio *rpm*

```
 1 #!/bin/bash
 2 # de-rpm.sh: Decomprime un archivio 'rpm'
 3
 4 : ${1?"Utilizzo: `basename $0` file_archivio"}
 5 # Bisogna specificare come argomento un archivio 'rpm'.
 6
 7
 8 TEMPFILE=$$.cpio              # File temporaneo con nome "unico".
 9                               # $$ è l'ID di processo dello script.
```

```
 10
 11 rpm2cpio < $1 > $TEMPFILE      # Converte l'archivio rpm in un
archivio cpio.
 12 cpio --make-directories -F $TEMPFILE -i  # Decomprime l'archivio
cpio.
 13 rm -f $TEMPFILE                          # Cancella l'archivio cpio.
 14
 15 exit 0
 16
 17 #  Esercizio:
 18 #  Aggiungete dei controlli per verificare se
 19 #+                       1) "file_archivio" esiste e
 20 #+                       2) è veramente un archivio rpm.
 21 #  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**.


Su alcuni sistemi commerciali UNIX, **zcat** è il sinonimo di **uncompress -c**, di conseguenza non funziona su file compressi con *gzip*.

Vedi anche Esempio 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**.


Le versioni più recenti di tar sono state aggiornate per supportare **bzip2**.

compress, **uncompress**

È 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**.


Il comando **znew** trasforma 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**.

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: Unix)
```

Esempio 12-27. Togliere i commenti da sorgenti C

```
  1 #!/bin/bash
  2 # strip-comment.sh: Toglie i commenti (/* COMMENTO */) in un
programma C.
  3
  4 E_NOARG=0
  5 E_ERR_ARG=66
  6 E_TIPO_FILE_ERRATO=67
  7
  8 if [ $# -eq "$E_NOARG" ]
  9 then
 10   echo "Utilizzo: `basename $0` file-C" >&2 #  Messaggio d'errore
allo stderr.
 11   exit $E_ERR_ARG
 12 fi
 13
 14 # Verifica il corretto tipo di file.
 15 tipo=`file $1 | awk '{ print $2, $3, $4, $5 }'`
 16 # "file $1" restituisce nome e tipo di file . . .
 17 # quindi awk rimuove il primo campo, il nome . . .
```

```
 18 # Dopo di che il risultato è posto nella variabile "tipo".
 19 tipo_corretto="ASCII C program text"
 20
 21 if [ "$tipo" != "$tipo_corretto" ]
 22 then
 23   echo
 24   echo "Questo script funziona solo su file sorgenti C."
 25   echo
 26   exit $E_TIPO_FILE_ERRATO
 27 fi
 28
 29
 30 # Script sed piuttosto criptico:
 31 #--------
 32 sed '
 33 /^\/\*/d
 34 /.*\/\*/d
 35 ' $1
 36 #--------
 37 # Facile da capire, se dedicate diverse ore ad imparare i fondamenti
di sed.
 38
 39
 40 #  È necessario aggiungere ancora una riga allo script sed per
trattare
 41 #+ quei casi in cui una riga di codice è seguita da un commento.
 42 #  Questo viene lasciato come esercizio (niente affatto banale).
 43
 44 #  Ancora, il codice precedente cancella le righe con un "*/" o
"/*", che non
 45 #+ è un risultato desiderabile.
 46
 47 exit 0
 48
 49
 50 # ----------------------------------------------------------------
----------
 51 #  Il codice oltre la linea non viene eseguito a causa del
precedente 'exit 0'.
 52
 53 # Stephane Chazelas suggerisce la seguente alternativa:
 54
 55 utilizzo() {
 56   echo "Utilizzo: `basename $0` file-C" >&2
 57   exit 1
 58 }
 59
 60 STRANO=`echo -n -e '\377'`   # oppure STRANO=$'\377'
 61 [[ $# -eq 1 ]] || utilizzo
 62 case `file "$1"` in
 63   *"C program text"*) sed -e "s%/\*%${STRANO}%g;s%\*/%${STRANO}%g"
"$1" \
 64   | tr '\377\n' '\n\377' \
 65   | sed -ne 'p;n' \
 66   | tr -d '\n' | tr '\377' '\n';;
 67   *) utilizzo;;
 68 esac
 69
 70 #  Questo può ancora essere ingannato da occorrenze come:
 71 #  printf("/*");
 72 #  o
 73 #  /*  /* errato commento annidato */
```

```
 74 #
 75 #  Per poter gestire tutti i casi particolari (commenti in stringhe,
commenti
 76 #+ in una stringa in cui è presente \", \\" ...) l'unico modo è
scrivere un
 77 #+ parser C (usando, forse, lex o yacc?).
 78
 79 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
```

Esempio 12-28. Esplorare `/usr/X11R6/bin`

```
  1 #!/bin/bash
  2
  3 # Cosa sono tutti quei misteriosi eseguibili in /usr/X11R6/bin?
  4
  5 DIRECTORY="/usr/X11R6/bin"
  6 # Provate anche "/bin", "/usr/bin", "/usr/local/bin", ecc.
  7
  8 for file in $DIRECTORY/*
  9 do
 10   whatis `basename $file` # Visualizza le informazione sugli
eseguibili.
 11 done
 12
 13 exit 0
 14 # Potreste desiderare di redirigere l'output di questo script, così:
 15 # ./what.sh >>whatis.db
 16 # o visualizzarne una pagina alla volta allo stdout,
 17 # ./what.sh | less
```

Vedi anche Esempio 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 Esempio 10-7 e Esempio 10-9.

Esempio 12-29. Un comando *strings* "migliorato"

```
 1 #!/bin/bash
 2 # wstrings.sh: "word-strings" (comando "strings" migliorato)
 3 #
 4 #  Questo script filtra l'output di "strings" confrontandolo
 5 #+ con un file dizionario.
 6 #  In questo modo viene eliminato efficacemente il superfluo,
 7 #+ restituendo solamente le parole riconosciute.
 8
```

```
 9 # ===========================================================
10 #       Verifica Standard del/degli Argomento/i dello Script
11 ARG=1
12 E_ERR_ARG=65
13 E_NOFILE=66
14
15 if [ $# -ne $ARG ]
16 then
17   echo "Utilizzo: `basename $0` nomefile"
18   exit $E_ERR_ARG
19 fi
20
21 if [ ! -f "$1" ]                       # Verifica l'esistenza del
file.
22 then
23     echo "Il file \"$1\" non esiste."
24     exit $E_NOFILE
25 fi
26 # ===========================================================
27
28
29 LUNMINSTR=3                             #  Lunghezza minima della
stringa.
30 DIZIONARIO=/usr/share/dict/linux.words #  File dizionario.
31                                        #  Può essere specificato un
file
32                                        #+ dizionario diverso purché
33                                        #+ di una parola per riga.
34
35
36 elenco=`strings "$nome_file" | tr A-Z a-z | tr '[:space:]' Z | \
37 tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
38
39 #  Modifica l'output del comando 'strings' mediante diversi passaggi
a 'tr'.
40 #  "tr A-Z a-z"  trasforma le lettere maiuscole in minuscole.
41 #  "tr '[:space:]' Z"  trasforma gli spazi in Z.
42 #  "tr -cs '[:alpha:]' Z"  trasforma i caratteri non alfabetici in
Z,
43 #+ riducendo ad una sola le Z multiple consecutive.
44 #  "tr -s '\173-\377' Z"  trasforma tutti i caratteri oltre la 'z'
in Z,
45 #+ riducendo ad una sola le Z multiple consecutive, liberandoci così
di tutti i
46 #+ caratteri strani che la precedente istruzione non è riuscita a
trattare.
47 #  Infine, "tr Z ' '" trasforma tutte queste Z in spazi, che saranno
48 #+ considerati separatori di parole dal ciclo che segue.
49
50 #  Notate la tecnica di concatenare diversi 'tr',
51 #+ ma con argomenti e/o opzioni differenti ad ogni passaggio.
52
53
54 for parola in $elenco                  #  Importante:
55                                        #  non bisogna usare $elenco
col quoting.
56                                        #  "$elenco" non funziona.
57                                        #  Perché?
58 do
59
60   lunstr=${#parola}                    #  Lunghezza della stringa.
61   if [ "$lunstr" -lt "$LUNMINSTR" ]    #  Salta le stringhe con meno
```

```
 62                                        #+ di 3 caratteri.
 63   then
 64     continue
 65   fi
 66
 67   grep -Fw $parola "$DIZIONARIO"       #  Cerca solo le parole
complete.
 68        ^^^                             #  Opzioni "stringhe Fisse" e
 69                                        #+ "parole (words) intere".
 70
 71 done
 72
 73
 74 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**.

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.

```
  1 patch -p1 <file-patch
  2 # Prende tutte le modifiche elencate in 'file-patch'
  3 # e le applica ai file che sono specificati in "file-patch".
  4 # Questo esegue l'aggiornamento del pacchetto alla versione più
recente.
```

Patch del kernel:

```
  1 cd /usr/src
```

```
 2 gzip -cd patchXX.gz | patch -p0
 3 # Aggiornamento dei sorgenti del kernel usando 'patch'.
 4 # Dal file "README" della documentazione del kernel Linux,
 5 # di autore anonimo (Alan Cox?).
```

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
```

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, è difficile che questo comando venga impiegato 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.

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.

Esempio 12-30. Utilizzare cmp in uno script per confrontare due file

```
 1 #!/bin/bash
 2
 3 ARG=2  # Lo script si aspetta due argomenti.
 4 E_ERR_ARG=65
 5 E_NONLEGGIBILE=66
 6
 7 if [ $# -ne "$ARG" ]
 8 then
 9   echo "Utilizzo: `basename $0` file1 file2"
```

```
 10   exit $E_ERR_ARG
 11 fi
 12
 13 if [[ ! -r "$1" || ! -r "$2" ]]
 14 then
 15   echo "Entrambi i file, per essere confrontati, devono esistere ed
avere"
 16   echo "i permessi di lettura."
 17   exit $E_NONLEGGIBILE
 18 fi
 19
 20 cmp $1 $2 &> /dev/null  #  /dev/null elimina la visualizzazione del
 21                         #+ risultato del comando"cmp".
 22 #   cmp -s $1 $2  ottiene lo stesso risultato (opzione "-s" di
"cmp")
 23 #   Grazie  Anders Gustavsson per averlo evidenziato.
 24 #
 25 # Funziona anche con 'diff', vale a dire, diff $1 $2 &> /dev/null
 26
 27 if [ $? -eq 0 ]         # Verifica l'exit status del comando "cmp".
 28 then
 29   echo "Il file \"$1\" è identico al file \"$2\"."
 30 else
 31   echo "Il file \"$1\" è diverso dal file \"$2\"."
 32 fi
 33
 34 exit 0
```

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:

```
  1 echo "Utilizzo: `basename $0` arg1 arg2 ... argn"
```

dirname

Elimina **basename**, dal nome del file, visualizzando solamente il suo percorso.

basename e **dirname** possono operare su una stringa qualsiasi. Non è necessario che l'argomento si riferisca ad un file esistente e neanche essere il nome di un file (vedi Esempio A-8).

Esempio 12-31. basename e dirname

```
  1 #!/bin/bash
  2
  3 a=/home/bozo/daily-journal.txt
  4
  5 echo "Basename di /home/bozo/daily-journal.txt = `basename $a`"
  6 echo "Dirname di /home/bozo/daily-journal.txt = `dirname $a`"
  7 echo
  8 # Funzionano entrambe anche con la sola ~.
  9 echo "La mia cartella personale è `basename ~/`."
 10 echo "La directory home della mia cartella personale è `dirname
~/`."
 11
 12 exit 0
```

split, **csplit**

Utility per suddividere un file in porzioni di dimensioni minori. Sono solitamente impiegate 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.

Il comando **csplit** suddivide il file in base al *contesto*. La suddivisione viene eseguita nei punti in cui i modelli sono verificati.

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 **5** check**sum**).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
```

```
8babc97a6f62a4649716f4df8d61728f  -
```

Il comando **cksum** visualizza anche la dimensione, in byte, del suo riferimento. sia esso un file o lo `stdout`.

Il comando **md5sum** visualizza un trattino quando l'input proviene dallo `stdout`.

Esempio 12-32. Verificare l'integrità dei file

```
 1 #!/bin/bash
 2 # file-integrity.sh: Verifica se i file di una data directory
 3 #                    sono stati modificati senza autorizzazione.
 4
 5 E_DIR_ERRATA=70
 6 E_ERR_DBFILE=71
 7
 8 dbfile=File_record.md5
 9 # Nome del file che contiene le registrazioni (file database).
10
11
12 crea_database ()
13 {
14   echo ""$directory"" > "$dbfile"
15   # Scrive il nome della directory come prima riga di dbfile.
16   md5sum "$directory"/* >> "$dbfile"
17   # Accoda le checksum md5 e i nomi dei file.
18 }
19
20 verifica_database ()
21 {
22   local n=0
23   local nomefile
24   local checksum
25
26   # ------------------------------------------------------------ #
27   #  Questa verifica potrebbe anche non essere necessaria, ma è
28   #+ meglio essere pignoli che rischiare.
29
30   if [ ! -r "$dbfile" ]
31   then
32       echo "Impossibile leggere il database delle checksum!"
33       exit $E_ERR_DBFILE
34   fi
35   # ------------------------------------------------------------ #
36
37   while read record[n]
38   do
39
40     directory_verificata="${record[0]}"
41     if [ "$directory_verificata" != "$directory" ]
42     then
43       echo "Le directory non corrispondono!"
44       # E' stato indicato un nome di directory sbagliato.
45       exit $E_DIR_ERRATA
46     fi
47
48     if [ "$n" -gt 0 ]   # Non è il nome della directory.
49     then
50       nomefile[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
```

```
 51       #  md5sum scrive nel primo campo la checksum, nel
 52       #+ secondo il nome del file.
 53       checksum[n]=$( md5sum "${nomefile[n]}" )
 54
 55
 56       if [ "${record[n]}" = "${checksum[n]}" ]
 57       then
 58         echo "${nomefile[n]} non è stato modificato."
 59
 60       elif [ "`basename ${nomefile[n]}`" != "$dbfile" ]
 61              #  Salta il database delle checksum,
 62              #+ perchè cambia ad ogni invocazione dello script.
 63          #  ---
 64          #  Questo significa, purtroppo, che quando si esegue
 65          #+ lo script su $PWD, la coincidenza con il
 66          #+ file database delle checksum non viene rilevata.
 67          #  Esercizio: Risolvete questo problema.
 68         then
 69           echo "${nomefile[n]} : CHECKSUM ERRATA!"
 70           # Il file è stato modificato dall'ultima verifica.
 71       fi
 72
 73     fi
 74
 75
 76
 77     let "n+=1"
 78   done <"$dbfile"   # Legge il database delle checksum.
 79
 80 }
 81
 82 # ============================================================ #
 83 # main ()
 84
 85 if [ -z  "$1" ]
 86 then
 87   directory="$PWD"      #  Se non altrimenti specificata, usa la
 88 else                    #+ directory di lavoro corrente.
 89   directory="$1"
 90 fi
 91
 92 clear                   # Pulisce lo schermo.
 93 echo " In esecuzione il controllo dell'integrità dei file in
$directory"
 94 echo
 95
 96 # ------------------------------------------------------------------
- #
 97   if [ ! -r "$dbfile" ] # Occorre creare il database?
 98   then
 99       echo "Sto creando il database, \""$directory"/"$dbfile"\".";
echo
100       crea_database
101   fi
102 # ------------------------------------------------------------------
- #
103
104 verifica_database       # Esegue il lavoro di verifica.
105
106 echo
107
108 #  Sarebbe desiderabile redirigere lo stdout dello script in un
```

```
file,
109 #+ specialmente se la directory da verificare contiene molti file.
110
111 exit 0
112
113 #  Per una verifica d'integrità molto più approfondita,
114 #+ considerate l'impiego del pacchetto "Tripwire",
115 #+ http://sourceforge.net/projects/tripwire/.
```

Vedi anche Esempio 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 Esempio 12-50, ma esegue il compito in maniera più completa ed elegante.

Questa è una delle *fileutils* GNU.


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.

Esempio 12-33. Decodificare file

```
  1 #!/bin/bash
  2 #  Decodifica con uudecode tutti i file della directory di lavoro
corrente
  3 #+ cifrati con uuencode.
  4
  5 righe=35        # 35 righe di intestazione (molto generoso).
  6
  7 for File in *   # Verifica tutti i file presenti in $PWD.
  8 do
  9   ricerca1=`head -$righe $File | grep begin | wc -w`
 10   ricerca2=`tail -$righe $File | grep end | wc -w`
 11   #  Decodifica i file che hanno un "begin" nella parte iniziale e
un "end"
 12   #+ in quella finale.
 13   if [ "$ricerca1" -gt 0 ]
 14   then
 15       if [ "$ricerca2" -gt 0 ]
 16       then
 17             echo "Decodifico con uudecode - $File -"
 18             uudecode $File
```

```
 19     fi
 20   fi
 21 done
 22
 23 #  Notate che se si invoca questo script su se stesso, l'esecuzione
è ingannata
 24 #+ perché pensa di trovarsi in presenza di un file codificato con
uuencode,
 25 #+ poiché contiene sia "begin" che "end".
 26
 27 #  Esercizio:
 28 #  Modificate lo script per verificare in ogni file l'intestazione
di un
 29 #+ newsgroup, saltando al file successivo nel caso non venga
trovata.
 30
 31 exit 0
```

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. [2] 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 realizzata proprio dall'autore del libro (vedi Esempio A-5).

Miscellanea

mktemp

Crea un file temporaneo con nome "unico".

```
  1 PREFISSO=nomefile
  2 tempfile=`mktemp $PREFISSO.XXXXXX`
  3 #                          ^^^^^^ Occorrono almeno 6 posti per
  4 #+                                il suffisso del nome del file.
  5 echo "nome del file temporaneo = $tempfile"
  6 # nome del file temporaneo = nomefile.QA2ZpY
  7 #                            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.

Note

[1] **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.

[2] 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

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 l'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

Visualizza 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

Domain **I**nformation **G**roper. Simile a **nslookup**, esegue una "risoluzione del nome del server" Internet. Può essere eseguito sia in modalità interattiva che non, vale a dire in uno script.

Alcune interessanti opzioni di **dig** sono: `+time=N` per impostare la temporizzazione della ricerca a *N* secondi, `+nofail` per far proseguire l'interrogazione dei server finché non si sia ottenuta una risposta e `-x` per effettuare una risoluzione inversa.

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
```

Esempio 12-34. Verificare un dominio di spam

```
  1 #! /bin/bash
  2 # is-spammer.sh: Identificare i domini di spam
  3
  4 # $Id: is-spammer, v 1.4 2004/09/01 19:37:52 mszick Exp $
  5 # La riga precedente indica l'ID del RCS.
  6 #
  7 #  È la versione semplificata dello script "is_spammer.bash
  8 #+ presente nell'appendice Script Aggiuntivi.
  9
 10 # is-spammer <nome.dominio>
 11
 12 # Viene usato il programma esterno 'dig'
 13 # Provato con la versione 9.2.4rc5
 14
 15 # Uso di funzioni.
 16 # Utilizzo di IFS per il controllo delle stringhe da assegnare agli
array.
 17 # Fa persino qualcosa di utile: controlla le blacklist dei server e-
mail.
 18
 19 # Si usa il nome.dominio presente nell'URL:
 20 # http://www.veramente_ottimo.spammer.biz/tutto_il_resto_ignorato
 21 #                              ^^^^^^^^^^
 22 # Oppure il nome.domainio dell'indirizzo e-mail:
 23 # Offerta_Strabiliante@spammer.biz
 24 #                      ^^^^^^^^^^
 25 # come unico argomento dello script.
 26 #(PS: occorre essere connessi ad internet)
 27 #
 28 # Concludendo, in base ai due esempi precedenti, questo script si
invoca con:
 29 #       is-spammer.sh spammer.biz
 30
 31
 32 # Spaziatura == :Spazio:Tabulazione:Line Feed:A_capo:
 33 SPZ_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'
 34
 35 # Nessuna spaziatura == Line Feed:A_capo
 36 No_SPZ=$'\x0A'$'\x0D'
 37
 38 # Separatore di campo per gli indirizzi ip puntati
 39 IND_IFS=${No_SPZ}'.'
 40
 41 # Recupera la registrazione del testo del dns
 42 # rec_testo <codice_errore> <server>
 43 rec_testo() {
 44
 45     # verifica $1 per l'assegnamento delle stringhe delimitate dai
punti
 46     local -a dns
 47     IFS=$IND_IFS
 48     dns=( $1 )
 49     IFS=$SPZ_IFS
 50     if [ "${dns[0]}" == '127' ]
 51     then
 52         # controlla se vi è una spiegazione
 53         echo $(dig +short $2 -t txt)
```

```
 54     fi
 55 }
 56
 57 # Recupera l'indirizzo dns
 58 # rec_idr <dns_inv> <server>
 59 rec_idr() {
 60     local risposta
 61     local server
 62     local causa
 63
 64     server=${1}${2}
 65     risposta=$( dig +short ${server} )
 66
 67     # se la risposta contiene un codice d'errore . . .
 68     if [ ${#risposta} -gt 6 ]
 69     then
 70         causa=$(rec_testo ${risposta} ${server} )
 71         causa=${causa:-${risposta}}
 72     fi
 73     echo ${causa:-' non in blacklist.'}
 74 }
 75
 76 # Si deve risalire all'indirizzo IP partendo dal nome di dominio.
 77 echo "Recupero l'indirizzo di: "$1
 78 ip_idr=$(dig +short $1)
 79 risposta_dns=${ip_idr:-' nessuna risposta '}
 80 echo ' Indirizzo: '${risposta_dns}
 81
 82 # Una risposta valida deve essere formata da almeno 4 cifre e 3
punti.
 83 if [ ${#ip_idr} -gt 6 ]
 84 then
 85     echo
 86     declare richiesta
 87
 88     # Controllo per l'assegnamento delle stringhe tra i punti.
 89     declare -a dns
 90     IFS=$IND_IFS
 91     dns=( ${ip_idr} )
 92     IFS=$SPZ_IFS
 93
 94     # Riordina gli ottetti nella sequenza adatta ad una
interrogazione dns.
 95     dns_inv="${dns[3]}"'.'"${dns[2]}"'.'"${dns[1]}"'.'"${dns[0]}"'.'
 96
 97 # Controlla su: http://www.spamhaus.org (Tradizionale, ben
mantenuto)
 98     echo -n 'spamhaus.org dice: '
 99     echo $(rec_idr ${dns_inv} 'sbl-xbl.spamhaus.org')
100
101 # Controlla su: http://ordb.org (Server aperti di instradamento e-
mail)
102     echo -n '   ordb.org  dice: '
103     echo $(rec_idr ${dns_inv} 'relays.ordb.org')
104
105 # Controlla su: http://www.spamcop.net/ (Qui si possono segnalare
gli spammer)
106     echo -n ' spamcop.net dice: '
107     echo $(rec_idr ${dns_inv} 'bl.spamcop.net')
108
109 # # # altre operazioni di blacklist # # #
110
```

```
111 # Controlla su: http://cbl.abuseat.org.
112     echo -n ' abuseat.org dice: '
113     echo $(rec_idr ${dns_inv} 'cbl.abuseat.org')
114
115 # Controlla su: http://dsbl.org/usage (Server vari di instradamento
e-mail)
116     echo
117     echo 'Elenchi di server distibuiti'
118     echo -n '       list.dsbl.org dice: '
119     echo $(rec_idr ${dns_inv} 'list.dsbl.org')
120
121     echo -n '   multihop.dsbl.org dice: '
122     echo $(rec_idr ${dns_inv} 'multihop.dsbl.org')
123
124     echo -n 'unconfirmed.dsbl.org dice: '
125     echo $(rec_idr ${dns_inv} 'unconfirmed.dsbl.org')
126
127 else
128     echo
129     echo 'Indirizzo inutilizzabile.'
130 fi
131
132 exit 0
133
134 # Esercizi:
135 # --------
136
137 # 1) Verificate gli argomenti passati allo script, in caso d'errore
138 #    l'esecuzione deve terminare con un messaggio appropriato.
139
140 # 2) Controllate l'avvenuta connessione prima dell'invocazione dello
script,
141 #    in caso contrario terminate con un appropriato messaggio
d'errore.
142
143 # 3) Sostituite la "codifica" dei server BHL* con delle variabili
generiche.
144
145 # 4) Impostate una temporizzazione per lo script usando l'opzione
"+time="
146      del comando 'dig'.
147
148 # * Black Hole Lists - Elenchi dei server di instradamento e-mail
aperti che,
149 #+  come tali, sono utilizzati dagli spammer [N.d.T.].
```

Per un'ancor più elaborata versione dello script precedente, vedi Esempio A-26.

traceroute

Traccia il percorso intrapreso dai pacchetti inviati ad un host remoto. Questo comando funziona su una LAN, una 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 un'altra macchina, 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 particolare server *whois* dev'essere interrogato. Vedi Esempio 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
 Office: 2355 Clown St., 543-1234
 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. [1]

chfn

Modifica le informazioni rivelate dal comando **finger**.

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 Esempio 17-6, Esempio A-5 ed Esempio A-14).

uucp

UNIX to UNIX copy - 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.

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.

```
 1 wget -p http://www.xyz23.com/file01.html
 2 wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
```

Esempio 12-35. Ottenere una quotazione di borsa

```
 1 #!/bin/bash
 2 # quote-fetch.sh: Scarica una quotazione di borsa.
 3
 4
 5 E_NOPARAM=66
 6
 7 if [ -z "$1" ]  # Si deve specificare il titolo (sigla) da cercare.
 8   then echo "Utilizzo: `basename $0` codice_titolo"
 9   exit $E_NOPARAM
10 fi
11
12 codice_titolo=$1
13
14 suffisso_file=.html
15 # Cerca un file HTML, per cui bisogna usare un nome appropriato.
16 URL='http://finance.yahoo.com/q?s='
17 # Servizio finanziario di Yahoo, con suffisso di ricerca del titolo.
18
19 # -----------------------------------------------------------------
20 wget -O ${codice_titolo}${suffisso_file} "${URL}${codice_titolo}"
21 # -----------------------------------------------------------------
22
23 exit $?
24
25 # Esercizi:
26 # --------
27 #
28 # 1) Aggiungete una verifica che confermi all'utente che esegue lo
script
29 #    l'avvenuto collegamento.
30 #    (Suggerimento: confrontate l'output di 'ps -ax' con "ppp" o
"connect."
31 #
32 # 2) Modificate lo script per scaricare il bollettino metereologico
locale,
33 #+   fornendo come argomento il codice di avviamento postale.
```

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.

```
 1 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.

Esempio 12-36. Uso di ssh

```
  1 #!/bin/bash
  2 # remote.bash: Uso di ssh.
  3
  4 # Esempio di Michael Zick.
  5 # Usato con il consenso dell'autore.
  6
  7
  8 #   Presupposti:
  9 #   -----------
 10 #   il df-2 non dev'esere stato impegnato ( '2>/dev/null' ).
 11 #   ssh/sshd presumono che lo stderr ('2') verrà visualizzato
all'utente.
 12 #
 13 #   sshd deve essere in esecuzione sulla macchina.
 14 #   Probabilmente questa è la situazione per qualsiasi distribuzione
'standard',
 15 #+  e senza aver fatto qualche strana impostazione di ssh-keygen.
 16
 17 # Provate ssh da riga di comando sulla vostra macchina:
 18 #
 19 # $ ssh $HOSTNAME
 20 # Se non sono state fatte impostazioni ulteriori, vi verrà chiesta
la password.
 21 #   inserite la password
 22 #   quindi  $ exit
 23 #
 24 # Ha funzionato? In questo caso siete pronti per un altro po' di
divertimento.
 25
 26 # Provate ssh come utente 'root':
 27 #
 28 #   $  ssh -l root $HOSTNAME
 29 #   Quando vi verrà chiesta la password, inserite quella di root,
non la vostra.
 30 #          Last login: Tue Aug 10 20:25:49 2004 from
```

```
localhost.localdomain
 31 #   Dopo di che 'exit'.
 32
 33 #  I comandi precedenti forniscono una shell interattiva.
 34 #  È possibile impostare sshd in modalità: 'comando singolo',
 35 #+ ma questo va oltre lo scopo dell'esempio.
 36 #  L'unica cosa da notare è: che quello che segue funziona
 37 #+ in modalità 'comando singolo'.
 38
 39
 40 # Il fondamentale comando di visualizzazione allo stdout (locale).
 41
 42 ls -l
 43
 44 # E ora lo stesso comando su una macchina remota.
 45 # Se desiderate, potete passare 'USERNAME' 'HOSTNAME' diversi:
 46 USER=${USERNAME:-$(whoami)}
 47 HOST=${HOSTNAME:-$(hostname)}
 48
 49 #  Ora eseguiamo la precedente riga di comando su un host remoto,
 50 #+ la trasmissione è totalmente criptata.
 51
 52 ssh -l ${USER} ${HOST} " ls -l "
 53
 54 #  Il risultato atteso è: l'elenco dei file della directory home
dell'utente
 55 #+ presente sulla macchina remota.
 56 #  Se volete vedere delle differenze, eseguite lo script da una
qualsiasi directory
 57 #+ diversa dalla vostra directory home.
 58
 59 #  In altre parole, il comando Bash viene passato come stringa tra
apici
 60 #+ alla shell remota, che lo esegue sulla macchina remota.
 61 #  In questo caso sshd esegue  ' bash -c "ls -l" '   per conto
vostro.
 62
 63 #  Per informazioni su argomenti quali il non dover inserire la
 64 #+ password/passphrase ad ogni riga di comando, vedi
 65 #+    man ssh
 66 #+    man ssh-keygen
 67 #+    man sshd_config.
 68
 69 exit 0
```

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.

Posta

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.

Esempio 12-37. Uno script che si auto-invia

```
  1 #!/bin/sh
  2 # self-mailer.sh: Script che si auto-invia
  3
  4 adr=${1:-`whoami`}    # Imposta l'utente corrente come predefinito,
se
  5                       #+ non altrimenti specificato.
  6 #  Digitando 'self-mailer.sh wiseguy@superdupergenius.com'
  7 #+ questo script viene inviato a quel destinatario.
  8 #  Il solo 'self-mailer.sh' (senza argomento) invia lo script alla
  9 #+ persona che l'ha invocato, per esempio, bozo@localhost.localdomain
 10 #
 11 #  Per i dettagli sul costrutto  ${parametro:-default}, vedi la
sezione
 12 #+ "Sostituzione di Parametro" del capitolo "Variabili Riviste".
 13
 14 #
================================================================================
 15   cat $0 | mail -s " Lo script \"`basename $0`\" si è auto-inviato."
"$adr"
 16 #
================================================================================
 17
 18 # ------------------------------------------------
 19 #  Saluti dallo script che si auto-invia.
 20 #  Una persona maliziosa ha eseguito questo script,
 21 #+ che ne ha provocato l'invio a te. Apparentemente,
 22 #+ certa gente non ha niente di meglio da fare
 23 #+ con il proprio tempo.
 24 # ------------------------------------------------
 25
 26 echo "Il `date`, lo script \"`basename $0`\" è stato inviato a
"$adr"."
 27
 28 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.

Note

[1] 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.

12.7. Comandi di controllo del 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 Esempio 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 è in grado 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 UNIX molto ben collaudata, e che 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)
```

Esempio 12-38. Rata mensile di un mutuo

```
  1 #!/bin/bash
  2 # monthlypmt.sh: Calcola la rata mensile di un mutuo (prestito).
  3
  4
  5 #  Questa è una modifica del codice del pacchetto "mcalc" (mortgage
calculator),
  6 #+ di Jeff Schmidt e Mendel Cooper (vostro devotissimo, autore di
  7 #+ questo documento).
```

```
  8 #  http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz
[15k]
  9
 10 echo
 11 echo "Dato il capitale, il tasso d'interesse e la durata del mutuo,"
 12 echo "calcola la rata di rimborso mensile."
 13
 14 denominatore=1.0
 15
 16 echo
 17 echo -n "Inserisci il capitale (senza i punti di separazione)"
 18 read capitale
 19 echo -n "Inserisci il tasso d'interesse (percentuale)" # Se 12%
inserisci "12",
 20                                                      #+ non ".12".
 21 read t_interesse
 22 echo -n "Inserisci la durata (mesi)"
 23 read durata
 24
 25
 26  t_interesse=$(echo "scale=9; $t_interesse/100.0" | bc) #  Lo
converte
 27                                                       #+ in
decimale.
 28                   # "scale" determina il numero delle cifre
decimali.
 29
 30
 31  tasso_interesse=$(echo "scale=9; $t_interesse/12 + 1.0" | bc)
 32
 33
 34  numeratore=$(echo "scale=9; $capitale*$tasso_interesse^$durata" |
bc)
 35
 36  echo; echo "Siate pazienti. È necessario un po' di tempo."
 37
 38  let "mesi = $durata - 1"
 39 #
==========================================================================
 40  for ((x=$mesi; x > 0; x--))
 41  do
 42     den=$(echo "scale=9; $tasso_interesse^$x" | bc)
 43     denominatore=$(echo "scale=9; $denominatore+$den" | bc)
 44     #  denominatore = $(($denominatore + $den"))
 45  done
 46 # ------------------------------------------------------------------
--
 47 #  Rick Boivie ha indicato un'implementazione più efficiente del
 48 #+ ciclo precedente che riduce di 2/3 il tempo di calcolo.
 49
 50 # for ((x=1; x <= $mesi; x++))
 51 # do
 52 #   denominatore=$(echo "scale=9; $denominatore * $tasso_interesse +
1" | bc)
 53 # done
 54
 55
 56 #  Dopo di che se n'è uscito con un'alternativa ancor più
efficiente, una che
 57 #+ abbatte il tempo di esecuzione di circa il 95%!
 58
 59 # denominatore=`{
```

```
 60 #  echo "scale=9; denominatore=$denominatore;
tasso_interesse=$tasso_interesse"
 61 #  for ((x=1; x <= $mesi; x++))
 62 #  do
 63 #       echo 'denominatore = denominatore * tasso_interesse + 1'
 64 #  done
 65 #  echo 'denominatore'
 66 #  } | bc`       #  Ha inserito il 'ciclo for' all'interno di una
 67                  #+ sostituzione di comando.
 68
 69 #
=================================================================
 70
 71  # let "rata = $numeratore/$denominatore"
 72  rata=$(echo "scale=2; $numeratore/$denominatore" | bc)
 73  # Vengono usate due cifre decimali per i centesimi di Euro.
 74
 75  echo
 76  echo "rata mensile = Euro $rata"
 77  echo
 78
 79
 80  exit 0
 81
 82  # Esercizi:
 83  #   1) Filtrate l'input per consentire l'inserimento del capitale
con i
 84  #      punti di separazione.
 85  #   2) Filtrate l'input per consentire l'inserimento del tasso
 86  #      d'interesse sia in forma percentuale che decimale.
 87  #   3) Se siete veramente ambiziosi, implementate lo script per
visualizzare
 88  #      il piano d'ammortamento completo.
 89
```

Esempio 12-39. Conversione di base

```
  1 #!/bin/bash

2 ##############################################################################
########
  3 # Shellscript:  base.sh - visualizza un numero in basi differenti
(Bourne Shell)
  4 # Autore     :  Heiner Steven (heiner.steven@odn.de)
  5 # Data       :  07-03-95
  6 # Categoria  :  Desktop
  7 # $Id        :  base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
  8 # ==> La riga precedente rappresenta l'ID RCS.

9 ##############################################################################
########
 10 # Descrizione
 11 #
 12 # Changes
 13 # 21-03-95 stv        fixed error occuring with 0xb as input (0.2)

14 ##############################################################################
#########
 15
 16 # ==> Utilizzato in questo documento con il permesso dell'autore dello
script.
```

```
 17 # ==> Commenti aggiunti dall'autore del libro.
 18
 19 NOARG=65
 20 NP=`basename "$0"`                                # Nome del programma
 21 VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`    # ==> VER=1.2
 22
 23 Utilizzo () {
 24     echo "$NP - visualizza un numero in basi diverse, $VER (stv '95)
 25 utilizzo: $NP [numero ...]
 26
 27 Se non viene fornito alcun numero, questi vengono letti dallo standard
input.
 28 Un numero può essere
 29     binario (base 2)               inizia con 0b (es. 0b1100)
 30     ottale (base 8)                inizia con 0  (es. 014)
 31     esadecimale (base 16)          inizia con 0x (es. 0xc)
 32     decimale                       negli altri casi (es. 12)" >&2
 33     exit $NOARG
 34 }   # ==> Funzione per la visualizzazione del messaggio di utilizzo.
 35
 36 Msg () {
 37     for i   # ==> manca in [lista].
 38     do echo "$NP: $i" >&2
 39     done
 40 }
 41
 42 Fatale () { Msg "$@"; exit 66; }
 43
 44 VisualizzaBasi () {
 45     # Determina la base del numero
 46     for i      # ==> manca in [lista] ...
 47     do         # ==> perciò opera sugli argomenti forniti da riga di
comando.
 48       case "$i" in
 49             0b*)                ibase=2;;        # binario
 50             0x*|[a-f]*|[A-F]*)  ibase=16;;       # esadecimale
 51             0*)                 ibase=8;;        # ottale
 52             [1-9]*)             ibase=10;;       # decimale
 53             *)
 54
 55                 Msg "$i numero non valido - ignorato"
 56                 continue;;
 57       esac
 58       #  Toglie il prefisso, converte le cifre esadecimali in
caratteri
 59       #+ maiuscoli (è richiesto da bc)
 60       numero=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
 61       # ==> Si usano i ":" come separatori per sed, al posto della
"/".
 62
 63       # Converte il numero in decimale
 64       dec=`echo "ibase=$ibase; $numero" | bc`
 65       # ==> 'bc' è l'utility di calcolo.
 66       case "$dec" in
 67       [0-9]*)        ;;                       # numero ok
 68       *)             continue;;               # errore: ignora
 69       esac
 70
 71       # Visualizza tutte le conversioni su un'unica riga.
 72       # ==> 'here document' fornisce una lista di comandi a 'bc'.
 73       echo `bc <<!
 74            obase=16; "esa="; $dec
```

```
 75            obase=10; "dec="; $dec
 76            obase=8;  "ott="; $dec
 77            obase=2;  "bin="; $dec
 78 !
 79       ` | sed -e 's: :        :g'
 80
 81     done
 82 }
 83
 84 while [ $# -gt 0 ]
 85 # ==>  "Ciclo while" che qui si rivela veramente necessario
 86 # ==>+ poiché, in ogni caso, o si esce dal ciclo
 87 # ==>+ oppure lo script termina.
 88 # ==> (Grazie, Paulo Marcel Coelho Aragao.)
 89 do
 90     case "$1" in
 91             --)        shift; break;;
 92             -h)        Utilizzo;;            # ==> Messaggio di aiuto.
 93             -*)        Utilizzo;;
 94              *)        break;;               # primo numero
 95     esac   # ==> Sarebbe utile un'ulteriore verifica d'errore per un
input
 96            #+    non consentito.
 97     shift
 98 done
 99
100 if [ $# -gt 0 ]
101 then
102     VisualizzaBasi "$@"
103 else                                        # legge dallo stdin
104     while read riga
105     do
106         VisualizzaBasi $riga
107     done
108 fi
109
110
111 exit 0
```

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**.

```
 1 variabile=`bc << STRINGA_LIMITE
 2 opzioni
 3 enunciati
 4 operazioni
 5 STRINGA_LIMITE
 6 `
 7
 8 ...oppure...
 9
10
11 variabile=$(bc << STRINGA_LIMITE
12 opzioni
13 enunciati
14 operazioni
15 STRINGA_LIMITE
16 )
```

Esempio 12-40. Invocare bc usando un "here document"

```
 1 #!/bin/bash
 2 # Invocare 'bc' usando la sostituzione di comando
 3 # in abbinamento con un 'here document'.
 4
 5
 6 var1=`bc << EOF
 7 18.33 * 19.78
 8 EOF
 9 `
10 echo $var1       # 362.56
11
12
13 #  $( ... ) anche questa notazione va bene.
14 v1=23.53
15 v2=17.881
16 v3=83.501
17 v4=171.63
18
19 var2=$(bc << EOF
20 scale = 4
21 a = ( $v1 + $v2 )
22 b = ( $v3 * $v4 )
23 a * b + 15.35
24 EOF
25 )
26 echo $var2       # 593487.8452
27
28
29 var3=$(bc -l << EOF
30 scale = 9
31 s ( 1.7 )
32 EOF
33 )
34 # Restituisce il seno di 1.7 radianti.
35 # L'opzione "-l" richiama la libreria matematica di 'bc'.
36 echo $var3       # .991664810
37
38
39 # Ora proviamolo in una funzione...
40 ip=              # Dichiarazione di variabile globale.
41 ipotenusa ()     # Calcola l'ipotenusa di un triangolo rettangolo.
42 {
43 ip=$(bc -l << EOF
44 scale = 9
45 sqrt ( $1 * $1 + $2 * $2 )
46 EOF
47 )
48 #  Sfortunatamente, non si può avere un valore di ritorno in virgola
mobile
49 #+ da una funzione Bash.
50 }
51
52 ipotenusa 3.68 7.31
53 echo "ipotenusa = $ip"     # 8.184039344
54
55
56 exit 0
```

Esempio 12-41. Calcolare il pi greco

```
  1 #!/bin/bash
  2 # cannon.sh: Approssimare il PI a cannonate.
  3
  4 # È un esempio molto semplice di una simulazione "Monte Carlo", un
modello
  5 #+ matematico di un evento reale, utilizzando i numeri pseudocasuali
per
  6 #+ simulare la probabilità dell'urna.
  7
  8 #  Consideriamo un appezzamento di terreno perfettamente quadrato,
di 10000
  9 #+ unità di lato.
 10 #  Questo terreno ha, al centro, un lago perfettamente circolare con
un
 11 #+ diametro di 10000 unità.
 12 #  L'appezzamento è praticamente tutta acqua, tranne i quattro
angoli
 13 #+ (Immaginatelo come un quadrato con inscritto un cerchio).
 14 #
 15 #  Spariamo delle palle con un vecchio cannone sul terreno quadrato.
Tutti i
 16 #+ proiettili cadranno in qualche parte dell'appezzamento, o nel
lago o negli
 17 #+ angoli emersi.
 18 #  Poiché il lago occupa la maggior parte dell'area del terreno, la
maggior
 19 #+ parte dei proiettili CADRA' nell'acqua.
 20 #  Solo pochi COLPIRANNO il terreno ai quattro angoli
dell'appezzamento.
 21 #
 22 #  Se le cannonate sparate saranno sufficientemente casuali, senza
aver
 23 #+ mirato, allora il rapporto tra le palle CADUTE IN ACQUA ed il
totale degli
 24 #+ spari approssimerà il valore di PI/4.
 25 #
 26 #  La spiegazione sta nel fatto che il cannone spara solo al
quadrante superiore
 27 #+ destro del quadrato, vale a dire, il 1° Quadrante del piano di
assi
 28 #+ cartesiani. (La precedente spiegazione era una semplificazione.)
 29 #
 30 #  Teoricamente, più alto è il numero delle cannonate, maggiore
 31 #+ sarà l'approssimazione.
 32 #  Tuttavia, uno script di shell, in confronto ad un linguaggio
compilato che
 33 #+ dispone delle funzione matematiche in virgola mobile, richiede un
po' di
 34 #+ compromessi.
 35 #  Sfortunatamente, questo fatto tende a diminuire la precisione
della
 36 #+ simulazione.
 37
 38
 39 DIMENSIONE=10000 #  Lunghezza dei lati dell'appezzamento di terreno.
 40                 #  Imposta anche il valore massimo degli interi
 41                 #+ casuali generati.
 42
 43 MAXSPARI=1000    #  Numero delle cannonate.
 44                 #  Sarebbe stato meglio 10000 o più, ma avrebbe
 45                 #+ richiesto troppo tempo.
 46                 #
```

```
 47 PMULTIPL=4.0     # Fattore di scala per approssimare PI.
 48
 49 genera_casuale ()
 50 {
 51 SEME=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
 52 RANDOM=$SEME                              #  Dallo script di esempio
 53                                           #+ "seeding-random.sh".
 54 let "rnum = $RANDOM % $DIMENSIONE"     #  Intervallo inferiore a
10000.
 55 echo $rnum
 56 }
 57
 58 distanza=        # Dichiarazione di variabile globale.
 59 ipotenusa ()     # Calcola l'ipotenusa di un triangolo rettangolo.
 60 {                # Dall'esempio "alt-bc.sh".
 61 distanza=$(bc -l <<EOF
 62 scale = 0
 63 sqrt ( $1 * $1 + $2 * $2 )
 64 EOF
 65 )
 66 #  Impostando "scale" a zero il risultato viene troncato (vengono
eliminati i
 67 #+ decimali), un compromesso necessario in questo script.
 68 #  Purtroppo. questo diminuisce la precisione della simulazione.
 69 }
 70
 71
 72 # main() {
 73
 74 # Inizializzazione variabili.
 75 spari=0
 76 splash=0
 77 terra=0
 78 Pi=0
 79
 80 while [ "$spari" -lt  "$MAXSPARI" ]       #  Ciclo principale.
 81 do
 82
 83   xCoord=$(genera_casuale)                #  Determina le
 84                                           #+ coordinate casuali X e
Y.
 85   yCoord=$(genera_casuale)
 86   ipotenusa $xCoord $yCoord               #  Ipotenusa del triangolo
 87                                           #+ rettangolo = distanza.
 88   ((spari++))
 89
 90   printf "#%4d   " $spari
 91   printf "Xc = %4d  " $xCoord
 92   printf "Yc = %4d  " $yCoord
 93   printf "Distanza = %5d  " $distanza     #  Distanza dal centro del
lago,
 94                                           #+ origine degli assi,
 95                                           #+ coordinate 0,0.
 96
 97   if [ "$distanza" -le "$DIMENSIONE" ]
 98   then
 99       echo -n "SPLASH!  "
100       ((splash++))
101   else
102       echo -n "TERRENO! "
103       ((terra++))
104   fi
```

```
105
106   Pi=$(echo "scale=9; $PMULTIPL*$splash/$spari" | bc)
107   # Moltiplica il rapporto per 4.0.
108   echo -n "PI ~ $Pi"
109   echo
110
111 done
112
113 echo
114 echo "Dopo $spari cannonate, $Pi sembra approssimare PI."
115 #  Tende ad essere un po' più alto . . .
116 #  Probabilmente a causa degli arrotondamenti e dell'imperfetta
casualità di
117 #+ $RANDOM.
118 echo
119
120 # }
121
122 exit 0
123
124 #  Qualcuno potrebbe ben chiedersi se uno script di shell sia
appropriato per
125 #+ un'applicazione così complessa e ad alto impiego di risorse
qual'è una
126 #+ simulazione.
127 #
128 #  Esistono almeno due giustificazioni.
129 #  1) Come prova concettuale: per dimostrare che può essere fatto.
130 #  2) Per prototipizzare e verificare gli algoritmi prima della
131 #+    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" - notazione polacca inversa). 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.

Esempio 12-42. Convertire un numero decimale in esadecimale

```
  1 #!/bin/bash
  2 # hexconvert.sh: Converte un numero decimale in esadecimale.
  3
  4 BASE=16     # Esadecimale.
  5
  6 if [ -z "$1" ]
  7 then
  8   echo "Utilizzo: $0 numero"
  9   exit $E_ERR_ARG
 10   # È necessario un argomento da riga di comando.
 11 fi
 12 # Esercizio: aggiungete un'ulteriore verifica di validità
dell'argomento.
 13
 14
 15 esacvt ()
 16 {
 17 if [ -z "$1" ]
```

```
 18 then
 19   echo 0
 20   return    #  "Restituisce" 0 se non è stato passato nessun
argomento alla
 21             #+ funzione.
 22 fi
 23
 24 echo ""$1" "$BASE" o p" | dc
 25 #                 "o" imposta la radice (base numerica) dell'output.
 26 #                   "p" visualizza la parte alta dello stack.
 27 # Vedi 'man dc' per le altre opzioni.
 28 return
 29 }
 30
 31 esacvt "$1"
 32
 33 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.

Esempio 12-43. Fattorizzazione

```
  1 #!/bin/bash
  2 # factr.sh: Fattorizza un numero
  3
  4 MIN=2       # Non funzionerà con con un numero inferiore a questo.
  5 E_ERR_ARG=65
  6 E_INFERIORE=66
  7
  8 if [ -z $1 ]
  9 then
 10   echo "Utilizzo: $0 numero"
 11   exit $E_ERR_ARG
 12 fi
 13
 14 if [ "$1" -lt "$MIN" ]
 15 then
 16   echo "Il numero da fattorizzare deve essere $MIN o maggiore."
 17   exit $E_INFERIORE
 18 fi
 19
 20 #  Esercizio: Aggiungete una verifica di tipo (per rifiutare un
argomento
 21 #+ diverso da un intero).
 22
 23 echo "Fattori primi di $1:"
 24 # ------------------------------------------------------------------
---------
 25 echo
"$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2"\
 26 | dc
 27 # ------------------------------------------------------------------
---------
 28 # La precedente riga di codice è stata scritta da Michel Charpentier
 29 # <charpov@cs.unh.edu>.
 30 # Usata con il permesso dell'autore (grazie).
 31
 32 exit 0
```

awk

Un altro modo ancora per eseguire calcoli in virgola mobile in uno script, è l'impiego delle funzioni matematiche built-in di awk in uno shell wrapper.

Esempio 12-44. Calcolo dell'ipotenusa di un triangolo

```
  1 #!/bin/bash
  2 # hypotenuse.sh: Calcola l'"ipotenusa" di un triangolo rettangolo.
  3 #               ( radice quadrata della somma dei quadrati dei
cateti)
  4
  5 ARG=2               # Lo script ha bisogno che gli vengano passati
i cateti
  6                     #+ del triangolo.
  7 E_ERR_ARG=65        # Numero di argomenti errato.
  8
  9 if [ $# -ne "$ARG" ] # Verifica il numero degli argomenti.
 10 then
 11   echo "Utilizzo: `basename $0` cateto_1 cateto_2"
 12   exit $E_ERR_ARG
 13 fi
 14
 15
 16 SCRIPTAWK=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
 17 #            comando/i / parametri passati ad awk
 18
 19
 20 echo -n "Ipotenusa di $1 e $2 = "
 21 echo $1 $2 | awk "$SCRIPTAWK"
 22
 23 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.

Esempio 12-45. Utilizzo di seq per generare gli argomenti di un ciclo

```
  1 #!/bin/bash
  2 # Uso di "seq"
  3
  4 echo
  5
  6 for a in `seq 80`  # oppure  for a in $( seq 80 )
  7 #  Uguale a   for a in 1 2 3 4 5 ... 80   (si risparmia molta
digitazione!).
  8 #+ Si potrebbe anche usare 'jot' (se presente nel sistema).
  9 do
 10   echo -n "$a "
 11 done      # 1 2 3 4 5 ... 80
 12 #  Esempio dell'uso dell'output di un comando per generare la
[lista] di un
 13 #+ ciclo "for".
 14
 15 echo; echo
 16
 17
 18 CONTO=80  # Sì, 'seq' può avere un parametro.
 19
 20 for a in `seq $CONTO`  # o   for a in $( seq $CONTO )
 21 do
 22   echo -n "$a "
 23 done      # 1 2 3 4 5 ... 80
 24
 25 echo; echo
 26
 27 INIZIO=75
 28 FINE=80
 29
 30 for a in `seq $INIZIO $FINE`
 31 #  Fornendo due argomenti "seq" inizia il conteggio partendo dal
primo e
 32 #+ continua fino a raggiungere il secondo.
 33 do
 34   echo -n "$a "
 35 done      # 75 76 77 78 79 80
 36
 37 echo; echo
 38
 39 INIZIO=45
 40 INTERVALLO=5
 41 FINE=80
 42
 43 for a in `seq $INIZIO $INTERVALLO $FINE`
 44 #  Fornendo tre argomenti "seq" inizia il conteggio partendo dal
primo, usa il
 45 #+ secondo come passo (incremento) e continua fino a raggiungere il
terzo.
 46 do
 47   echo -n "$a "
 48 done      # 45 50 55 60 65 70 75 80
 49
```

```
 50 echo; echo
 51
 52 exit 0
```

getopt

getopt verifica le opzioni, precedute da un trattino, passate da riga di comando. È il comando esterno corrispondente al builtin di Bash getopts. **getopt**, usato con l'opzione `-l`, permette la gestione delle opzioni estese nonchè il riordino dei parametri.

Esempio 12-46. Utilizzo di getopt per analizzare le opzioni passate da riga di comando

```
  1 #!/bin/bash
  2 # Usare getopt
  3
  4 # Provate ad invocare lo script nei modi seguenti:
  5 #   sh ex33a.sh -a
  6 #   sh ex33a.sh -abc
  7 #   sh ex33a.sh -a -b -c
  8 #   sh ex33a.sh -d
  9 #   sh ex33a.sh -dXYZ
 10 #   sh ex33a.sh -d XYZ
 11 #   sh ex33a.sh -abcd
 12 #   sh ex33a.sh -abcdZ
 13 #   sh ex33a.sh -z
 14 #   sh ex33a.sh a
 15 # Spiegate i risultati di ognuna delle precedenti esecuzioni.
 16
 17 E_ERR_OPZ=65
 18
 19 if [ "$#" -eq 0 ]
 20 then   # Lo script richiede almeno un argomento da riga di comando.
 21   echo "Utilizzo $0 -[opzioni a,b,c]"
 22   exit $E_ERR_OPZ
 23 fi
 24
 25 set -- `getopt "abcd:" "$@"`
 26 # Imposta come parametri posizionali gli argomenti passati da riga
di comando.
 27 # Cosa succede se si usa "$*" invece di "$@"?
 28
 29 while [ ! -z "$1" ]
 30 do
 31   case "$1" in
 32    -a) echo "Opzione \"a\"";;
 33    -b) echo "Opzione \"b\"";;
 34    -c) echo "Opzione \"c\"";;
 35    -d) echo "Opzione \"d\" $2";;
 36     *) break;;
 37   esac
 38
 39   shift
 40 done
 41
 42 #  Solitamente in uno script è meglio usare il builtin 'getopts',
 43 #+ piuttosto che 'getopt'.
 44 #  Vedi "ex33.sh".
 45
 46 exit 0
```

run-parts

Il comando **run-parts** [1] 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 cron 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!).

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. [2] 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 per la formattazione del testo, 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
=================================
```

```
1 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 misterioso comando crea una *named pipe*, un *buffer first-in-first-out* temporaneo, per il trasferimento di dati tra processi. [3] Tipicamente, un processo scrive nel FIFO e un altro vi legge. Vedi Esempio 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, [4] 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.

Esempio 12-47. Uno script che copia sè stesso

```
 1 #!/bin/bash
 2 # self-copy.sh
 3
 4 # Questo script copia se stesso.
 5
 6 suffisso_file=copia
 7
 8 dd if=$0 of=$0.$suffisso_file 2>/dev/null
 9 # Sopprime i messaggi di dd:  ^^^^^^^^^^^
10
11 exit $?
```

Esempio 12-48. Esercitarsi con dd

```
 1 #!/bin/bash
 2 # exercising-dd.sh
 3
 4 # Script di Stephane Chazelas.
 5 # Con qualche modifica eseguita dall'autore del libro.
 6
 7 file_input=$0   # Questo script.
 8 file_output=log.txt
 9 n=3
10 p=5
11
12 dd if=$file_input of=$file_output bs=1 skip=$((n-1)) count=$((p-
n+1)) 2> /dev/null
13 # Toglie i caratteri da n a p dallo script.
14
15 # ------------------------------------------------------
16
17 echo -n "ciao mondo" | dd cbs=1 conv=unblock 2> /dev/null
18 # Visualizza "ciao mondo" verticalmente.
19
20 exit 0
```

Per dimostrare quanto versatile sia **dd**, lo usiamo per catturare i tasti premuti.

Esempio 12-49. Intercettare i tasti premuti

```
 1 #!/bin/bash
 2 # Intercetta i tasti premuti senza dover premere anche INVIO.
 3
 4
 5 tastidapremere=4                      # Numero di tasti da
catturare.
 6
 7
 8 precedenti_impostazioni_tty=$(stty -g) #  Salva le precedenti
 9                                       #+ impostazioni del
terminale.
10
11 echo "Premi $tastidapremere tasti."
12 stty -icanon -echo                    # Disabilita la modalità
canonica.
13                                       # Disabilita l'eco locale.
14 tasti=$(dd bs=1 count=$tastidapremere 2> /dev/null)
```

```
 15 # 'dd' usa lo stdin, se non viene specificato "if".
 16
 17 stty "$precedenti_impostazioni_tty"    # Ripristina le precedenti
impostazioni.
 18
 19 echo "Hai premuto i tasti \"$tasti\"."
 20
 21 # Grazie, S.C. per la dimostrazione.
 22 exit 0
```

Il comando **dd** può eseguire un accesso casuale in un flusso di dati.

```
  1 echo -n . | dd bs=1 seek=4 of=file conv=notrunc
  2 #  L'opzione "conv=notrunc" significa che il file di output non
verrà troncato.
  3
  4 # 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 (Esempio 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 (Esempio 29-2) e di ramdisk (Esempio 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**.

Esempio 12-50. Cancellare in modo sicuro un file

```
  1 #!/bin/bash
  2 # blot-out.sh: Cancella ogni traccia del file.
  3
  4 #  Questo script sovrascrive il file di riferimento alternativamente
con byte
  5 #+ casuali e con zeri, prima della cancellazione finale.
  6 #  Dopo di che, anche un esame diretto dei settori del disco non
riuscirà a
  7 #+ rivelare i dati originari del file.
  8
  9 PASSI=7          #  Numero di sovrascritture.
 10                  #  Aumentando questo valore si rallenta
l'esecuzione dello
 11                  #+ script, specialmente con i file di grandi
dimensioni.
 12 DIMBLOCCO=1      #  L'I/O con /dev/urandom richiede di specificare
la dimensione
 13                  #+ del blocco, altrimenti si ottengono risultati
```

```
strani.
 14 E_ERR_ARG=70     #  Codice d'uscita per errori generici.
 15 E_FILE_NON_TROVATO=71
 16 E_CAMBIO_IDEA=72
 17
 18 if [ -z "$1" ]   # Nessun nome di file specificato.
 19 then
 20   echo "Utilizzo: `basename $0` nomefile"
 21   exit $E_ERR_ARG
 22 fi
 23
 24 file=$1
 25
 26 if [ ! -e "$file" ]
 27 then
 28   echo "Il file \"$file\" non è stato trovato."
 29   exit $E_FILE_NON_TROVATO
 30 fi
 31
 32 echo; echo -n "Sei assolutamente sicuro di voler cancellare
\"$file\" (s/n)? "
 33 read risposta
 34 case "$risposta" in
 35 [nN]) echo "Hai cambiato idea, vero?"
 36       exit $E_CAMBIO_IDEA
 37       ;;
 38 *)    echo "Cancellazione del file \"$file\".";;
 39 esac
 40
 41
 42 dim_file=$(ls -l "$file" | awk '{print $5}')  #  Il 5° campo è la
dimensione
 43                                               #+ del file.
 44 conta_passi=1
 45
 46 chmod u+w "$file"  # Consente di sovrascrivere/cancellare il file.
 47
 48 echo
 49
 50 while [ "$conta-passi" -le "$PASSI" ]
 51 do
 52   echo "Passaggio nr.$conta_passi"
 53   sync             # Scarica i buffer.
 54   dd if=/dev/urandom of=$file bs=$DIMBLOCCO count=$dim_file
 55                    # Sovrascrive con byte casuali.
 56   sync             # Scarica ancora i buffer.
 57   dd if=/dev/zero of=$file bs=$DIMBLOCCO count=$dim_file
 58                    # Sovrascrive con zeri.
 59   sync             # Scarica ancora una volta i buffer.
 60   let "conta_passi += 1"
 61   echo
 62 done
 63
 64
 65 rm -f $file    # Infine, cancella il file.
 66 sync           # Scarica i buffer un'ultima volta.
 67
 68 echo "Il file \"$file\" è stato cancellato."; echo
 69
 70
 71 exit 0
 72
```

```
 73 #  È un metodo abbastanza sicuro, sebbene lento ed inefficiente, per
rendere un
 74 #+ file completamente "irriconoscibile".
 75 #  Il comando "shred", che fa parte del pacchetto GNU "fileutils",
esegue lo
 76 #+ stesso lavoro, ma in maniera molto più efficiente.
 77
 78 #  La cancellazione non può essere "annullata" né il file recuperato
con i
 79 #+ metodi consueti.
 80 #  Tuttavia . . .
 81 #+ questo semplice metodo probabilmente *non* resisterebbe a
un'analisi forense.
 82
 83 #  Questo script potrebbe non funzionare correttamente con un file
system journaled.
 84 #  Esercizio: risolvete questo problema.
 85
 86
 87
 88 #  Il pacchetto per la cancellazione sicura di file "wipe" di Tom
Vier esegue
 89 #+ un lavoro molto più completo di quanto non faccia questo semplice
script.
 90 #     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2
 91
 92 #  Per un'analisi approfondita sull'argomento della cancellazione
sicura dei
 93 #+ file, vedi lo studio di Peter Gutmann,
 94 #+     "Secure Deletion of Data From Magnetic and Solid-State
Memory".
 95 #
http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
```

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 Esempio 9-27 e Esempio 12-13.

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 informazioni su 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.

```
1 random000=$(mcookie)
```

Naturalmente, uno script potrebbe utilizzare per lo stesso scopo md5.

```
1 # Genera una checksum md5 dello script stesso.
2 random001=`md5sum $0 | awk '{print $1}'`
3 # Usa 'awk' per eliminare il nome del file.
```

Il comando **mcookie** fornisce un altro metodo, ancora, per generare un nome di file "unico" .

Esempio 12-51. Generatore di nomi di file

```
 1 #!/bin/bash
 2 # tempfile-name.sh:  generatore di nomi di file temporanei
 3
 4 STR_BASE=`mcookie`   # magic cookie di 32-caratteri.
 5 POS=11               # Posizione arbitraria nella stringa magic
cookie.
 6 LUN=5                # Ottiene $LUN caratteri consecutivi.
 7
 8 prefisso=temp        #  È, dopo tutto, un file "temporaneo".
 9                      #  Per una maggiore "unicità", generate il
prefisso del
10                      #+ nome del file usando lo stesso metodo del
11                      #+ suffisso, di seguito.
12
13 suffisso=${STR_BASE:POS:LUN}
14                      #  Estrae una stringa di 5-caratteri, iniziando
dall'11a
15                      #+ posizione.
16
17 nomefile_temp=$prefisso.$suffisso
18                      # Crea il nome del file.
19
20 echo "Nome del file temporaneo = "$nomefile_temp""
21
22 # sh tempfile-name.sh
23 # Nome del file temporaneo = temp.e19ea
24
25 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.

Esempio 12-52. Convertire i metri in miglia

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

```
 2 # unit-conversion.sh
 3
 4
 5 converte_unità ()  # Vuole come argomenti le unità da convertire.
 6 {
 7   cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')
 8   # Toglie tutto tranne il reale fattore di conversione.
 9   echo "$cf"
10 }
11
12 Unità1=miglia
13 Unità2=metri
14 fatt_conv =`converte_unità $Unità1 $Unità2`
15 quantità=3.73
16
17 risultato=$(echo $quantità*$fatt_conv | bc)
18
19 echo "Ci sono $risultato $Unità2 in $quantità $Unità1."
20
21 #  Cosa succede se vengono passate alla funzione unità di misura
22 #+ incompatibili, come "acri" e "miglia"?
23
24 exit 0
```

m4

Un tesoro nascosto, **m4** è un potente filtro per l'elaborazione di macro, [5] 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 vi è un bellissimo articolo su **m4** ed i suoi impieghi.

Esempio 12-53. Utilizzo di m4

```
 1 #!/bin/bash
 2 # m4.sh: Uso del processore di macro m4
 3
 4 # Stringhe
 5 stringa=abcdA01
 6 echo "len($stringa)" | m4                          # 7
 7 echo "substr($stringa,4)" | m4                     # A01
 8 echo "regexp($stringa,[0-1][0-1],\&Z)" | m4        # 01Z
 9
10 # Calcoli aritmetici
11 echo "incr(22)" | m4                               # 23
12 echo "eval(99 / 3)" | m4                           # 33
13
14 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*.


I diversi comportamenti dell'eseguibile devono essere definiti nel codice dell'eseguibile stesso, qualcosa di analogo al seguente script di shell:

```
1 case `basename $0` in
2 "nome1" ) fa_qualcosa;;
3 "nome2" ) fa_qualcosaltro;;
4 "nome3" ) fa_un_altra_cosa_ancora;;
5 *       ) azione_predefinita;;
6 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 Esempio 34-16.

sox

Il comando **sox**, ovvero "*so*und e*x*change", permette di ascoltare i file audio e anche di modificarne il formato.

Per esempio, **sox fileaudio.wav fileaudio.au** trasforma un file musicale dal formato WAV al formato AU (audio Sun).

Gli script di shell sono l'ideale per eseguire in modalità batch operazioni **sox** sui file audio.

Note

[1] In realtà si tratta dell'adattamento di uno script della distribuzione Debian GNU/Linux.

[2] Per *coda di stampa* si intende l'insieme dei job "in attesa" di essere stampati.

[3] Per un'eccellente disamina di quest'argomento vedi l'articolo di Andy Vaught, Introduction to Named Pipes, nel numero del Settembre 1997 di Linux Journal.

[4] 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.

```
  1 cat $file | dd conv=swab,ebcdic > $file_cifrato
  2 #  Codifica (lo rende inintelligibile).
  3 #  Si potrebbe anche fare lo switch dei byte (swab), per
rendere la cosa un po'
  4 #+ più oscura.
  5
  6 cat $file_cifrato | dd conv=swab,ascii > $file_testo
  7 # Decodifica.
```

[5] Una *macro* è una costante simbolica che si espande in un comando o in una serie di operazioni sui parametri.

Capitolo 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.

```
  1 chgrp --recursive dunderheads *.data
  2 #  Il gruppo "dunderheads" adesso è proprietario di tutti i
file"*.data"
  3 #+ 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.


Il comando **adduser** è il sinonimo di **useradd** nonché, di solito, un link simbolico ad esso.

usermod

Modifica l'account di un utente. La variazione può riguardare la password, il gruppo d'appartenenza, la data di scadenza ed altri attributi dell'account di un determinato utente. Con questo comando è possibile anche bloccare la password di un utente, con il risultato di disabilitare l'account dello stesso.

groupmod

Modifica gli attributi di un dato gruppo. Usando questo comando si può cambiare il nome del gruppo e/o il suo numero ID.

id

Il comando **id** elenca i reali ID utente e di gruppo dell'utente associato al processo 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
```


id mostra gli ID *effettivi* solo quando questi sono diversi da quelli *reali*.

Vedi anche Esempio 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
```


Mentre **logname** visualizza il nome dell'utente connesso, **whoami** fornisce il nome dell'utente collegato al processo corrente. Come si è appena visto, talvolta questi non coincidono.

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 Esempio 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.

```
 1 #!/bin/bash
 2
 3 # Alcuni comandi.
 4 sudo cp /root/secretfile /home/bozo/secret
 5 # 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.

Esempio 13-1. Impostare una nuova password

```
  1 #!/bin/bash
  2 #  setnew-password.sh: A solo scopo dimostrativo.
  3 #                      Non è una buona idea eseguire veramente
questo script.
  4 #  Deve essere eseguito da root.
  5
  6 UID_ROOT=0          # Root ha $UID 0.
  7 E_UTENTE_ERRATO=65  # Non root?
  8
  9 E_UTENTE_INESISTENTE=70
 10 SUCCESSO=0
 11
 12
 13 if [ "$UID" -ne "$UID_ROOT" ]
 14 then
 15   echo; echo "Solo root può eseguire lo script."; echo
 16   exit $E_UTENTE_ERRATO
 17 else
 18   echo
 19   echo "Root, dovresti saper far di meglio che non eseguire questo
script."
 20   echo "Anche gli utenti root hanno le loro giornate storte... "
 21   echo
 22 fi
 23
 24
 25 nomeutente=bozo
 26 NUOVAPASSWORD=violazione_sicurezza
 27
 28 # Controlla se l'utente bozo esiste.
 29 grep -q "$nomeutente" /etc/passwd
 30 if [ $? -ne $SUCCESSO ]
 31 then
 32   echo "L'utente $nomeutente non esiste."
 33   echo "Nessuna password modificata."
 34   exit $E_UTENTE_INESISTENTE
 35 fi
 36
 37 echo "$NUOVAPASSWORD" | passwd --stdin "$nomeutente"
 38 #  L'opzione '--stdin' di 'passwd' consente di
 39 #+ ottenere la nuova password dallo stdin (o da una pipe).
 40
 41 echo; echo "E' stata cambiata la password dell'utente $nomeutente!"
 42
 43 # E' pericoloso usare il comando 'passwd' in uno script.
 44
 45 exit 0
```

Le opzioni `-l`, `-u` e `-d` del comando **passwd** consentono di bloccare, sbloccare e cancellare la password di un utente. Solamente root può usare queste opzioni.

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 dei login effettuati da remoto.

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.

Esempio 13-2. Abilitare un carattere di cancellazione

```
  1 #!/bin/bash
  2 # erase.sh: Uso di "stty" per impostare un carattere di
cancellazione nella
  3 #+          lettura dell'input.
  4
  5 echo -n "Come ti chiami? "
  6 read nome                     #  Provate ad usare il tasto di
ritorno
  7                               #+ (backspace) per cancellare i
caratteri
  8                               #+ digitati. Problemi?.
  9 echo "Ti chiami $nome."
 10
 11 stty erase '#'                #  Imposta il carattere "hash" (#)
come
```

```
 12                                #+ carattere di cancellazione.
 13 echo -n "Come ti chiami? "
 14 read nome                      #  Usate # per cancellare l'ultimo
carattere
 15                                #+ digitato.
 16 echo "Ti chiami $nome."
 17
 18 # Attenzione: questa impostazione rimane anche dopo l'uscita dallo
script.
 19
 20 exit 0
```

Esempio 13-3. Password segreta: disabilitare la visualizzazione a terminale

```
  1 #!/bin/bash
  2
  3 echo
  4 echo -n "Introduci la password "
  5 read passwd
  6 echo "La password è $passwd"
  7 echo -n "Se qualcuno stesse sbirciando da dietro le vostre spalle,"
  8 echo "la password sarebbe compromessa."
  9
 10 echo && echo  # Due righe vuote con una "lista and".
 11
 12 stty -echo    # Disabilita la visualizzazione sullo schermo.
 13
 14 echo -n "Reimmetti la password "
 15 read passwd
 16 echo
 17 echo "La password è $passwd"
 18 echo
 19
 20 stty echo     # Ripristina la visualizzazione sullo schermo.
 21
 22 exit 0
```

Un uso creativo di **stty** è quello di rilevare i tasti premuti dall'utente (senza dover premere successivamente **INVIO**).

Esempio 13-4. Rilevamento dei tasti premuti

```
  1 #!/bin/bash
  2 # keypress.sh: Rileva i tasti premuti dall'utente ("tastiera
bollente").
  3
  4 echo
  5
  6 precedenti_impostazioni_tty=$(stty -g)  # Salva le precedenti
impostazioni.
  7 stty -icanon
  8 tasti=$(head -c1)             #  Oppure $(dd bs=1 count=1 2>
/dev/null)
  9                               #+ su sistemi non-GNU
 10
 11 echo
 12 echo "Hai premuto i tasti \""$tasti"\"."
 13 echo
 14
```

```
 15 stty "$precedenti_impostazioni_tty"     # Ripristina le precedenti
impostazioni.
 16
 17 # Grazie, Stephane Chazelas.
 18
 19 exit 0
```

Vedi anche Esempio 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. Tutti i tasti premuti vengono registrati in un buffer specifico per quel terminale. 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

setterm

Imposta alcuni attributi del terminale. Questo comando scrive una stringa nello `stdout` del proprio terminale con la quale modifica il comportamento del terminale stesso.

```
bash$ setterm -cursor off
bash$
```

setterm può essere usato in uno script per modificare le modalità: di visualizzazione di un testo allo `stdout`, anche se esistono certamente strumenti migliori per questo scopo.

```
 1 setterm -bold on
 2 echo ciao in grassetto
 3
 4 setterm -bold off
 5 echo ciao normale
```

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.

```
 1 # Dallo script /etc/pcmcia/serial:
 2
 3 IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
 4 setserial /dev/$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.

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, invia 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 Esempio 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!
```


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 generati durante la 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 Esempio 12-5). 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 Esempio 10-26.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Fornisce informazioni sui comandi precedentemente eseguiti, 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
```

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.

```
  1 #!/bin/bash
  2
  3 SERVER=$HOST                              # localhost.localdomain
(127.0.0.1).
  4 NUMERO_PORTA=25                           # porta SMTP.
  5
  6 nmap $SERVER | grep -w "$NUMERO_PORTA"  # Questa specifica porta è
aperta?
  7 #              grep -w verifica solamente la parola esatta,
  8 #+             così, per esempio, non verrà verificata la porta
1025.
  9
 10 exit 0
 11
 12 # 25/tcp     open        smtp
```

free

Mostra, in forma tabellare, l'utilizzo della memoria e della cache. Il suo output si presta molto bene 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 le tabelle di routing e le connessioni attive. Questa utility accede alle informazioni presenti in `/proc/net` (Capitolo 28). Vedi Esempio 28-3.

netstat -r equivale a route.

```
bash$ netstat
 Active Internet connections (w/o servers)
 Proto Recv-Q Send-Q Local Address           Foreign Address
State
 Active UNIX domain sockets (w/o servers)
 Proto RefCnt Flags       Type       State         I-Node Path
 unix  11     [ ]         DGRAM                    906    /dev/log
 unix  3      [ ]         STREAM     CONNECTED     4514   /tmp/.X11-
unix/X0
 unix  3      [ ]         STREAM     CONNECTED     4513
 . . .
```

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
```

Simili al comando **hostname** sono **domainname**, **dnsdomainname**, **nisdomainname** e **ypdomainname**. Questi possono essere usati per visualizzare o impostare il DNS di sistema o il nome di dominio NIS/YP. Anche diverse opzioni di **hostname** svolgono queste funzioni.

hostid

Visualizza un identificatore numerico esadecimale a 32 bit dell'host della macchina.

```
bash$ hostid
7f0100
```

Si presume che questo comando possa fornire un numero di serie "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, scritto da Sebastien Godard.

```
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**.

```
  1 logger Riscontrata un'instabilità nella connessione di rete alle
23:10, 21/05.
  2 # Ora eseguite 'tail /var/log/messages'.
```

Inserendo il comando **logger** in uno script è possibile scrivere informazioni di debugging in `/var/log/messages`.

```
  1 logger -t $0 -i Logging alla riga "$LINENO".
  2 # L'opzione "-t" specifica l'identificativo della registrazione di
logger
  3 # L'opzione "-i" registra l'ID di processo.
  4
  5 # tail /var/log/message
  6 # ...
  7 # 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 cron 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 Esempio 11-11 e Esempio 28-2).

```
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
```

Esempio 13-5. pidof aiuta ad terminare un processo

```
  1 #!/bin/bash
  2 # kill-process.sh
  3
  4 NESSUNPROCESSO=2
  5
  6 processo=xxxyyyzzz  # Si usa un processo inesistente.
  7 # Solo a scopo dimostrativo...
  8 # ... con questo script non si vuole terminare nessun processo in
esecuzione.
  9 #
 10 # Se però voleste, per esempio, usarlo per scollegarvi da Internet,
allora
 11 #     processo=pppd
 12
 13 t=`pidof $processo` # Cerca il pid (id di processo) di $processo.
 14 #  Il pid è necessario a 'kill' (non si può usare 'kill' con
 15 #+ il nome del programma).
 16
 17 if [ -z "$t" ]      # Se il processo non è presente, 'pidof'
restituisce null.
 18 then
 19   echo "Il processo $processo non è in esecuzione."
 20   echo "Non è stato terminato alcun processo."
 21   exit $NESSUNPROCESSO
 22 fi
 23
 24 kill $t             # Potrebbe servire 'kill -9' per un processo
testardo.
 25
 26 #  Qui sarebbe necessaria una verifica, per vedere se il processo ha
 27 #+ acconsentito ad essere terminato.
 28 #  Forse un altro " t=`pidof $processo` ".
 29
 30
 31 # L'intero script potrebbe essere sostituito da
 32 #    kill $(pidof -x nome_processo)
 33 # ma non sarebbe stato altrettanto istruttivo.
 34
 35 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, specialmente negli script che hanno come scopo quello di evitare, agli utenti non autorizzati, l'accesso ai servizi di sistema.

fuser si rivela un'applicazione importante nel momento in cui si devono inserire o rimuovere fisicamente dispositivi di memorizzazione, come i CD ROM o le memorie flash USB. Talvolta umount fallisce con il messaggio d'errore device is busy. Questo sta ad indicare che qualche utente e/o processo(i) hanno accesso a quel dispositivo. Un **fuser -um /nome_dispositivo** vi rivelerà il mistero, così che possiate terminare tutti i processi coinvolti.

```
bash$ umount /mnt/driveusb
umount: /mnt/driveusb: device is busy



bash$ fuser -um /dev/driveusb
/mnt/driveusb:        1772c(bozo)

bash$ kill -9 1772
bash$ umount /mnt/driveusb
```

cron

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`


Alcune distribuzioni Linux eseguono **crond**, la versione **cron** di Matthew Dillon.

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, si rende 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, ovvero 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.

```
bash$ ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)
```

ifconfig viene usato molto spesso in fase di boot per impostare le interfacce, o per disabilitarle in caso di riavvio.

```
  1 # Frammenti di codice dal file /etc/rc.d/init.d/network
  2
  3 # ...
  4
  5 # Controlla se la rete è attiva.
  6 [ ${NETWORKING} = "no" ] && exit 0
  7
  8 [ -x /sbin/ifconfig ] || exit 0
  9
 10 # ...
 11
 12 for i in $interfaces ; do
 13   if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
 14     action "L'interfaccia $i non è attiva: " ./ifdown $i boot
 15   fi
 16 #  L'opzione "-q" di "grep", che è una specifica GNU, significa
 17 #+ "quiet", cioè, non produce output.
 18 #  Quindi, redirigere l'output in /dev/null non è strettamente
necessario.
 19
 20 # ...
 21
 22 echo "Attualmente sono attivi questi dispositivi:"
 23 echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
 24 #                            ^^^^^
 25 #           si dovrebbe usare il quoting per evitare il globbing.
 26 #  Anche le forme seguenti vanno bene.
 27 #    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
 28 #    echo $(/sbin/ifconfig | sed -e 's/ .*//')
 29 #  Grazie, S.C. per i commenti aggiuntivi.
```

Vedi anche Esempio 30-6.

iwconfig

È il comando predisposto per la configurazione di una rete wireless. È l'equivalente wireless del precedente **ifconfig**, .

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 del 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.

```
1 mount -t iso9660 /dev/cdrom /mnt/cdrom
2 # Monta il CDROM
```

```
  3 mount /mnt/cdrom
  4 # 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]

Esempio 13-6. Verificare un'immagine CD

```
  1 # Da root...
  2
  3 mkdir /mnt/cdtest  # Prepara un punto di mount, nel caso non
esistesse.
  4
  5 mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Monta
l'immagine.
  6 #                  l'opzione "-o loop" equivale a "losetup
/dev/loop0"
  7 cd /mnt/cdtest     # Ora verifica l'immagine.
  8 ls -alR            # Elenca i file della directory.
  9                    # 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.

```
  1 umount /mnt/cdrom
  2 # Ora potete premere il tasto eject e rimuovere in tutta sicurezza
il disco.
```

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 Esempio 12-50) o quando le luci di casa incominciano a tremolare.

losetup

Imposta e configura i dispositivi di loopback.

Esempio 13-7. Creare un filesystem in un file

```
  1 DIMENSIONE=1000000  # 1 mega
  2
  3 head -c $DIMENSIONE < /dev/zero > file  #  Imposta il file alla
  4                                         #+ dimensione indicata.
  5 losetup /dev/loop0 file                 #  Lo imposta come
dispositivo
  6                                         #+ di loopback.
  7 mke2fs /dev/loop0                       # Crea il filesystem.
  8 mount -o loop /dev/loop0 /mnt           # Lo monta.
  9
 10 # 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.

Esempio 13-8. Aggiungere un nuovo hard disk

```
  1 #!/bin/bash
  2
  3 # Aggiunge un secondo hard disk al sistema.
  4 # Configurazione software. Si assume che l'hardware sia già montato
sul PC.
  5 # Da un articolo dell'autore di questo libro.
  6 # Pubblicato sul nr. 38 di "Linux Gazette",
http://www.linuxgazette.com.
  7
  8 ROOT_UID=0     # Lo script deve essere eseguito da root.
  9 E_NONROOT=67   # Errore d'uscita non-root.
 10
 11 if [ "$UID" -ne "$ROOT_UID" ]
 12 then
 13   echo "Devi essere root per eseguire questo script."
 14   exit $E_NONROOT
 15 fi
 16
 17 #  Da usare con estrema attenzione!
 18 #  Se qualcosa dovesse andare storto, potreste cancellare
irrimediabilmente
 19 #+ il filesystem corrente.
 20
 21
 22 NUOVODISCO=/dev/hdb        # Si assume che sia libero /dev/hdb.
Verificate!
 23 MOUNTPOINT=/mnt/nuovodisco # Oppure scegliete un altro punto di
montaggio.
 24
 25 fdisk $NUOVODISCO
 26 mke2fs -cv $NUOVODISCO1    #  Verifica i blocchi difettosi
```

```
visualizzando un
 27                            #+ output dettagliato.
 28 #  Nota:    /dev/hdb1, *non* /dev/hdb!
 29 mkdir $MOUNTPOINT
 30 chmod 777 $MOUNTPOINT     # Rende il nuovo disco accessibile a tutti
gli utenti.
 31
 32
 33 # Ora, una verifica...
 34 # mount -t ext2 /dev/hdb1 /mnt/nuovodisco
 35 # Provate a creare una directory.
 36 # Se l'operazione riesce, smontate la partizione e procedete.
 37
 38 # Passo finale:
 39 # Aggiungete la riga seguente in /etc/fstab.
 40 # /dev/hdb1  /mnt/nuovodisco  ext2  defaults  1 1
 41
 42 exit 0
```

Vedi anche Esempio 13-7 e Esempio 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.


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.


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!


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.

lsusb, **usbmodules**

Il comando **lsusb** elenca tutti i bus USB (Universal Serial Bus) e i dispositivi ad essi collegati.

Il comando **usbmodules** visualizza le informazioni sui moduli dei dispositivi USB collegati.

```
root# lsusb
Bus 001 Device 001: ID 0000:0000
 Device Descriptor:
   bLength                18
   bDescriptorType         1
   bcdUSB               1.00
   bDeviceClass            9 Hub
   bDeviceSubClass         0
   bDeviceProtocol         0
   bMaxPacketSize0         8
   idVendor           0x0000
   idProduct          0x0000
   . . .
```

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

Cambia 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). 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.


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). 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.

```
  1 nomeapplicazione=xyzip
  2 # L'applicazione "xyzip" ha creato il file lock
"/var/lock/xyzip.lock".
  3
  4 if [ -e "/var/lock/$nomeapplicazione.lock ]
  5
  6 then
  7   ...
```

mknod

Crea file di dispositivo a blocchi o a caratteri (potrebbe essere necessario per l'installazione di nuovo hardware sul sistema). L'utility **MAKEDEV** possiede tutte le funzionalità di **mknod** ed è più facile da usare.

MAKEDEV

Utility per la creazione di file di dispositivo. Deve essere eseguita da root e ci si deve trovare nella directory `/dev`.

```
root# ./MAKEDEV
```

È una specie di versione avanzata di **mknod**.

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.

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 su dispositivi di registrazione più vari, 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 Capitolo 27).


Un uso giudizioso di **ulimit** può proteggere il sistema contro una temibile *bomba fork*.

```
  1 #!/bin/bash
  2 # Script a solo scopo illustrativo.
  3 # L'esecuzione è a vostro rischio -- vi *bloccherà* il
sistema.
  4
  5 while true  #  Ciclo infinito.
  6 do
  7   $0 &      #  Lo script invoca se stesso . . .
  8             #+ genera il processo un numero infinito di
volte . . .
  9             #+ finché il sistema non si blocca a seguito
```

```
 10             #+ dell'esaurimento di tutte le risorse.
 11 done        #  Questo è il famigerato scenario
dell'"apprendista stregone".
 12
 13 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.

setquota

Imposta, da riga di comando, le quote disco di un utente o di un gruppo.

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 Capitolo 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]
```


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.

modinfo

Visualizza informazioni su un modulo caricabile.

```
bash$ modinfo hid
filename:    /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
 description: "USB HID support drivers"
 author:      "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
 license:     "GPL"
```

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.


In Bash e in altre shell derivate dalla Bourne, è possibile impostare le variabili nell'ambiente di un singolo comando.

```
  1 var1=valore1 var2=valore2 comandoXXX
  2 # $var1 e $var2 vengono impostate solo nell'ambiente di
'comandoXXX'.
```


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.

```
  1 #! /usr/bin/env perl
  2
  3 print "Questo script Perl verrà eseguito,\n";
  4 print "anche quando non sai dove si trova l'interprete
Perl.\n";
  5
```

```
  6 # Ottimo per la portabilità degli script su altre
piattaforme,
  7 # dove i binari Perl potrebbero non essere dove ci
aspettiamo.
  8 # 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`.

```
  1 watch -n 5 tail /var/log/messages
  2 #  Visualizza la parte finale del file di log di sistema
/var/log/messages
  3 #+ 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.

Esempio 13-9. killall, da `/etc/rc.d/init.d`

```
 1 #!/bin/sh
 2
 3 # --> I commenti aggiunti dall'autore del libro sono indicati con "# -->".
 4
 5 # --> Questo fa parte del pacchetto di script 'rc'
 6 # --> di Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
 7
 8 # --> Sembra che questo particolare script sia specifico di Red Hat
```

```
 9 # --> (potrebbe non essere presente in altre distribuzioni).
10
11 # Bring down all unneeded services that are still running (there shouldn't
12 # be any, so this is just a sanity check)
13
14 for i in /var/lock/subsys/*; do
15         # --> Ciclo standard for/in, ma poiché "do" è posto sulla stessa
riga,
16         # --> è necessario aggiungere il ";".
17         # Check if the script is there.
18         [ ! -f $i ] && continue
19         # --> Ecco un uso intelligente di una "lista and", equivale a:
20         # --> if [ ! -f "$i" ]; then continue
21
22         # Get the subsystem name.
23         subsys=${i#/var/lock/subsys/}
24         # --> Imposta la variabile, in questo caso, al nome del file.
25         # --> È l'equivalente esatto di subsys=`basename $i`.
26
27         # -->  Viene ricavato dal nome del file lock (se esiste un file lock
28         # -->+ che rappresenti la prova che il processo è in esecuzione).
29         # -->  Vedi la precedente voce "lockfile".
30
31
32         # Bring the subsystem down.
33         if [ -f /etc/rc.d/init.d/$subsys.init ]; then
34             /etc/rc.d/init.d/$subsys.init stop
35         else
36             /etc/rc.d/init.d/$subsys stop
37         # -->  Sospende i job ed i demoni in esecuzione.
38         # -->  E' da notare che "stop" è un parametro posizionale,
39         # -->+ non un builtin di shell.
40         fi
41 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 "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".

Note

[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, nel numero dell'Ottobre 1999 di Linux Journal.

[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.

Capitolo 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. [2]

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.

```
 1 nome_script=`basename $0`
 2 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.

```
 1 rm `cat nomefile`   # "nomefile" contiene l'elenco dei file da cancellare.
 2 #
 3 # S. C. fa notare che potrebbe ritornare l'errore "arg list too long".
 4 # Meglio              xargs rm -- < nomefile
 5 # ( -- serve nei casi in cui "nomefile" inizia con un "-" )
 6
 7 elenco_filetesto=`ls *.txt`
 8 #  La variabile contiene i nomi di tutti i file *.txt della directory
 9 #+ di lavoro corrente.
10 echo $elenco_filetesto
11
12 elenco_filetesto2=$(ls *.txt)  # Forma alternativa di sostituzione di
comando.
13 echo $elenco_filetesto2
14 # Stesso risultato.
15
16 # Un problema possibile, nell'inserire un elenco di file in un'unica
stringa,
17 # è che si potrebbe insinuare un ritorno a capo.
18 #
19 #  Un modo più sicuro per assegnare un elenco di file ad un parametro
20 #+ è usare un array.
21 #      shopt -s nullglob       # Se non viene trovato niente, il nome del
file
22 #+                              non viene espanso.
23 #      elenco_filetesto=( *.txt )
24 #
25 # Grazie, S.C.
```

La sostituzione di comando invoca una subshell.

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 'nome di directory con un carattere di a capo finale
'

cd 'nome di directory con un carattere di a capo finale
'

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.
```

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`
```

```
  2 echo $elenco_directory     # senza quoting
  3
  4 # Ci si potrebbe aspettare un elenco di directory ben ordinato.
  5
  6 # Invece, quello che si ottiene è:
  7 # total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
  8 # bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13
wi.sh
  9
 10 # I ritorni a capo sono scomparsi.
 11
 12
 13 echo "$elenco_directory"   # con il quoting
 14 # -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
 15 # -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
 16 # -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

La sostituzione di comando consente anche di impostare una variabile al contenuto di un file, sia usando la redirezione che con il comando cat.

```
  1 variabile1=`<file1`      #  Imposta "variabile1" al contenuto di "file1".
  2 variabile2=`cat file2`   #  Imposta "variabile2" al contenuto di "file2".
  3                          #  Questo, tuttavia, genera un nuovo processo,
quindi
  4                          #+ questa riga di codice viene eseguita più
  5                          #+ lentamente della precedente.
  6
  7 #  Nota:
  8 #  Le variabili potrebbero contenere degli spazi,
  9 #+ o addirittura (orrore) caratteri di controllo.
```

```
  1 #  Frammenti scelti dal file di sistema /etc/rc.d/rc.sysinit
  2 #+ (su un'installazione Red Hat Linux)
  3
  4
  5 if [ -f /fsckoptions ]; then
  6         fsckoptions=`cat /fsckoptions`
  7 ...
  8 fi
  9 #
 10 #
 11 if [ -e "/proc/ide/${disk[$device]}/media" ] ; \
 12          then hdmedia=`cat /proc/ide/${disk[$device]}/media`
 13 ...
 14 fi
 15 #
 16 #
 17 if [ ! -n "`uname -r | grep -- "-"`" ]; then
 18        ktag="`cat /proc/version`"
 19 ...
 20 fi
 21 #
 22 #
 23 if [ $usb = "1" ]; then
 24     sleep 5
 25     mouseoutput=`cat /proc/bus/usb/devices \
 26                   2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
 27     kbdoutput=`cat /proc/bus/usb/devices \
 28                2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
 29 ...
 30 fi
```

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.

Esempio 14-1. Stupid script tricks

```
  1 #!/bin/bash
  2 # stupid-script-tricks.sh: Gente, non eseguitelo!
  3 # Da "Stupid Script Tricks," Volume I.
  4
  5
  6 variabile_pericolosa=`cat /boot/vmlinuz`   # Il kernel Linux compresso.
  7
  8 echo "Lunghezza della stringa \$variabile_pericolosa =
${#variabile_pericolosa}"
  9 # Lunghezza della stringa $variabile_pericolosa = 794151
 10 # (Non dà lo stesso risultato di 'wc -c /boot/vmlinuz'.)
 11
 12 # echo "$variabile_pericolosa"
 13 # Non fatelo! Bloccherebbe l'esecuzione dello script.
 14
 15
 16 #  L'autore di questo documento vi ha informato sull'assoluta inutilità
delle
 17 #+ applicazioni che impostano una variabile al contenuto di un file
binario.
 18
 19 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.

Esempio 14-2. Generare una variabile da un ciclo

```
  1 #!/bin/bash
  2 # csubloop.sh: Impostazione di una variabile all'output di un ciclo.
  3
  4 variabile1=`for i in 1 2 3 4 5
  5 do
  6   echo -n "$i"                   #  In questo caso il comando 'echo' è
  7 done`                            #+  cruciale nella sostituzione di
comando.
  8
  9 echo "variabile1 = $variabile1"  # variabile1 = 12345
 10
 11
 12 i=0
 13 variabile2=`while [ "$i" -lt 10 ]
 14 do
 15   echo -n "$i"                   # Ancora, il necessario 'echo'.
 16   let "i += 1"                   # Incremento.
 17 done`
 18
 19 echo "variabile2 = $variabile2"  # variabile2 = 0123456789
```

```
20
21 #  E' dimostrato che è possibile inserire un ciclo
22 #+ in una dichiarazione di variabile.
23
24 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.

```
 1 #include <stdio.h>
 2
 3 /*  Programma C "Ciao, mondo"  */
 4
 5 int main()
 6 {
 7   printf( "Ciao, mondo." );
 8   return (0);
 9 }
```

```
bash$ gcc -o ciao ciao.c
```

```
 1 #!/bin/bash
 2 # hello.sh
 3
 4 saluti=`./ciao`
 5 echo $saluti
```

```
bash$ sh hello.sh
Ciao, mondo.
```

Nella sostituzione di comando, la forma **$(COMANDO)** ha preso il posto di quella con gli apostrofi inversi.

```
 1 output=$(sed -n /"$1"/p $file) # Dall'esempio "grp.sh".
 2
 3 # Impostazione di una variabile al contenuto di un file di testo.
 4 Contenuto_file1=$(cat $file1)
 5 Contenuto_file2=$(<$file2)     # Bash consente anche questa forma.
```

La sostituzione di comando nella forma **$(...)** gestisce la doppia barra inversa in modo diverso che non nella forma **`...`**.

```
bash$ echo `echo \\`


bash$ echo $(echo \\)
\
```

Esempi di sostituzione di comando negli script di shell:

1. Esempio 10-7
2. Esempio 10-26
3. Esempio 9-27
4. Esempio 12-3

5. Esempio 12-18
6. Esempio 12-15
7. Esempio 12-45
8. Esempio 10-13
9. Esempio 10-10
10. Esempio 12-27
11. Esempio 16-8
12. Esempio A-18
13. Esempio 28-2
14. Esempio 12-38
15. Esempio 12-39
16. Esempio 12-40

Note

[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.

[2] In senso tecnicamente più corretto, la *sostituzione di comando* estrae lo `stdout` di un comando riassegnandolo successivamente ad una variabile usando l'operatore =.

Capitolo 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)

```
  1 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 dalle doppie parentesi -- **((...))** e **$((...))** -- e anche dal convenientissimo costrutto **let**.

```
  1 z=$(($z+3))
  2 z=$((z+3))                                #  Corretto anche
così.
  3                                           #  All'interno delle
doppie
  4                                           #+ parentesi la
dereferenziazione
  5                                           #+ del parametro è
opzionale.
  6
  7 # $((ESPRESSIONE)) è un'espansione aritmetica. #  Non deve essere
confusa con
  8                                           #+ la sostituzione di
comando.
  9
 10
```

```
 11
 12 # Nelle doppie parentesi è possibile effettuare operazioni senza
assegnamento.
 13
 14   n=0
 15   echo "n = $n"                                  # n = 0
 16
 17   (( n += 1 ))                                   # Incremento.
 18 # (( $n += 1 )) non è corretto!
 19   echo "n = $n"                                  # n = 1
 20
 21
 22 let z=z+3
 23 let "z += 3"  #  Il quoting consente l'uso degli spazi.
 24              #  L'operatore 'let' in realtà esegue una valutazione
aritmetica,
 25              #+ piuttosto che un'espansione.
```

Esempi di espansione aritmetica negli script:

1. Esempio 12-9
2. Esempio 10-14
3. Esempio 26-1
4. Esempio 26-11
5. Esempio A-18

Capitolo 16. Redirezione I/O

Sommario

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 Esempio 3-1 e Esempio 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 Esempio 16-1).

```
1    OUTPUT_COMANDO >
2       # Redirige lo stdout in un file.
3       # Crea il file se non è presente, in caso contrario lo sovrascrive.
4
5       ls -lR > dir-albero.list
6       # Crea un file contente l'elenco dell'albero delle directory.
7
8    : > nomefile
```

```
  9       # Il > svuota il file "nomefile", lo riduce a dimensione zero.
 10       # Se il file non è presente, ne crea uno vuoto (come con 'touch').
 11       # I : servono da segnaposto e non producono alcun output.
 12
 13     > nomefile
 14       #  Il > svuota il file "nomefile", lo riduce a dimensione zero.
 15       #  Se il file non è presente, ne crea uno vuoto (come con 'touch').
 16       #  (Stesso risultato di ": >", visto prima, ma questo, con alcune
 17       #+ shell, non funziona.)
 18
 19    OUTPUT_COMANDO >>
 20       # Redirige lo stdout in un file.
 21       # Crea il file se non è presente, in caso contrario accoda l'output.
 22
 23
 24       #  Comandi di redirezione di riga singola
 25       #+ (hanno effetto solo sulla riga in cui si trovano):
 26       #-------------------------------------------------------------------
 27
 28     1>nomefile
 29       # Redirige lo stdout nel file "nomefile".
 30     1>>nomefile
 31       # Redirige e accoda lo stdout nel file "nomefile".
 32     2>nomefile
 33       # Redirige lo stderr nel file "nomefile".
 34     2>>nomefile
 35       # Redirige e accoda lo stderr nel file "nomefile".
 36     &>nomefile
 37       # Redirige sia lo stdout che lo stderr nel file "nomefile".
 38
 39       #==================================================================
 40       # Redirigere lo stdout una riga alla volta.
 41       FILELOG=script.log
 42
 43       echo "Questo enunciato viene inviato al file di log, \
 44            \"$FILELOG\"." 1>$FILELOG
 45       echo "Questo enunciato viene accodato in \"$FILELOG\"." 1>>$FILELOG
 46       echo "Anche questo enunciato viene accodato in \"$FILELOG\"."
1>>$FILELOG
 47       echo "Questo enunciato viene visualizzato allo \
 48             stdout e non comparirà in \"$FILELOG\"."
 49       #  Questi comandi di redirezione vengono automaticamente "annullati"
 50       #+ dopo ogni riga.
 51
 52
 53
 54       # Redirigere lo stderr una riga alla volta.
 55       FILEERRORI=script.err
 56
 57       comando_errato1 2>$FILEERRORI          #  Il messaggio d'errore viene
 58                                              #+ inviato in $FILEERRORI.
 59       comando_errato2 2>>$FILEERRORI         #  Il messaggio d'errore viene
 60                                              #+ accodato in $FILEERRORI.
 61       comando_errato3                        #  Il messaggio d'errore viene
 62                                              #+ visualizzato allo stderr,
 63                                              #+ e non comparirà in
$FILEERRORI.
 64       #  Questi comandi di redirezione vengono automaticamente "annullati"
 65       #+ dopo ogni riga.
 66       #==================================================================
 67
 68
```

```
 69
 70    2>&1
 71       # Redirige lo stderr allo stdout.
 72       # I messaggi d'errore vengono visualizzati a video, ma come stdout.
 73
 74     i>&j
 75       #  Redirige il descrittore di file i in j.
 76       #  Tutti gli output del file puntato da i vengono inviati al file
 77       #+ puntato da j.
 78
 79    >&j
 80       # Redirige, per default, il descrittore di file 1 (lo stdout) in j.
 81       # Tutti gli stdout vengono inviati al file puntato da j.
 82
 83     0< NOMEFILE
 84      < NOMEFILE
 85       # Riceve l'input da un file.
 86       # È il compagno di ">" e vengono spesso usati insieme.
 87       #
 88       # grep parola-da-cercare <nomefile
 89
 90
 91    [j]<>nomefile
 92       #  Apre il file "nomefile" in lettura e scrittura, e gli assegna il
 93       #+ descrittore di file "j".
 94       #  Se il file "nomefile" non esiste, lo crea.
 95       #  Se il descrittore di file "j" non viene specificato, si assume per
 96       #+ default il df 0, lo stdin.
 97       #
 98       #  Una sua applicazione è quella di scrivere in un punto specifico
 99       #+ all'interno di un file.
100       echo 1234567890 > File    #  Scrive la stringa in "File".
101       exec 3<> File             #  Apre "File" e gli assegna il df 3.
102       read -n 4 <&3             #  Legge solo 4 caratteri.
103       echo -n . >&3             #  Scrive il punto decimale dopo i
104                                 #+ caratteri letti.
105       exec 3>&-                 #  Chiude il df 3.
106       cat File                  #  ==> 1234.67890
107       # È l'accesso casuale, diamine.
108
109    |
110       # Pipe.
111       # Strumento generico per concatenare processi e comandi.
112       # Simile a ">", ma con effetti più generali.
113       # Utile per concatenare comandi, script, file e programmi.
114       cat *.txt | sort | uniq > file-finale
115       # Ordina gli output di tutti i file .txt e cancella le righe doppie,
116       # 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.

```
 1 comando < file-input > file-output
 2
 3 comando1 | comando2 | comando3 > file-output
```

Vedi Esempio 12-26 e Esempio A-16.

È possibile redirigere più flussi di output in un unico file.

```
  1 ls -yz >> comandi.log 2>&1
  2 #  Invia il risultato delle opzioni illegali "yz" di "ls" nel file
"comandi.log"
  3 #  Poiché lo stderr è stato rediretto nel file, in esso si troveranno anche
  4 #+ tutti i messaggi d'errore.
  5
  6 #  Notate, però, che la riga seguente *non* dà lo stesso risultato.
  7 ls -yz 2>&1 >> comandi.log
  8 #  Visualizza un messaggio d'errore e non scrive niente nel file.
  9
 10 #  Nella redirezione congiunta di stdout e stderr,
 11 #+ non è indifferente l'ordine dei comandi.
```

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.

```
  1 # Redirige solo lo stderr alla pipe.
  2
  3 exec 3>&1                           #  Salva il "valore" corrente dello
stdout.
  4 ls -l 2>&1 >&3 3>&- | grep bad 3>&- #  Chiude il df 3 per 'grep' (ma
  5                                     #+ non per'ls').
  6 #              ^^^^   ^^^^
  7 exec 3>&-                           #  Ora è chiuso anche per la parte
  8                                     #+ restante dello script.
  9
 10 # Grazie, S.C.
```

Per una più dettagliata introduzione alla redirezione I/O vedi Appendice E.

16.1. Uso di 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.

Esempio 16-1. Redirigere lo `stdin` usando exec

```
 1 #!/bin/bash
 2 # Redirigere lo stdin usando 'exec'.
 3
 4
 5 exec 6<&0          # Collega il descrittore di file nr.6 allo stdin.
 6                    # Salva lo stdin.
 7
 8 exec < file-dati   # lo stdin viene sostituito dal file "file-dati"
 9
10 read a1            # Legge la prima riga del file "file-dati".
11 read a2            # Legge la seconda riga del file "file-dati."
12
13 echo
14 echo "Le righe lette dal file."
15 echo "------------------------"
16 echo $a1
17 echo $a2
18
19 echo; echo; echo
20
21 exec 0<&6 6<&-
22 #  È stato ripristinato lo stdin dal df nr.6, dov'era stato salvato,
23 #+ e chiuso il df nr.6 ( 6<&- ) per renderlo disponibile per un altro
processo.
24 #
25 # <&6 6<&-    anche questo va bene.
26
27 echo -n "Immetti dei dati  "
28 read b1  # Ora "read" funziona come al solito, leggendo dallo stdin.
29 echo "Input letto dallo stdin."
30 echo "------------------------"
31 echo "b1 = $b1"
32
33 echo
34
35 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.

Esempio 16-2. Redirigere lo `stdout` utilizzando exec

```
 1 #!/bin/bash
 2 # reassign-stdout.sh
 3
 4 FILELOG=filelog.txt
 5
 6 exec 6>&1           # Collega il descrittore di file nr.6 allo stdout.
 7                     # Salva lo stdout.
 8
 9 exec > $FILELOG     # stdout sostituito dal file "filelog.txt".
10
11 # ----------------------------------------------------------------- #
```

```
 12 #  Tutti i risultati dei comandi inclusi in questo blocco di codice vengono
 13 #+ inviati al file $FILELOG.
 14
 15 echo -n "File di log: "
 16 date
 17 echo "------------------------------------"
 18 echo
 19
 20 echo "Output del comando \"ls -al\""
 21 echo
 22 ls -al
 23 echo; echo
 24 echo "Output del comando \"df\""
 25 echo
 26 df
 27
 28 # ----------------------------------------------------------------------- #
 29
 30 exec 1>&6 6>&-    # Ripristina lo stdout e chiude il descrittore di file
nr.6.
 31
 32 echo
 33 echo "== ripristinato lo stdout alla funzionalità di default  == "
 34 echo
 35 ls -al
 36 echo
 37
 38 exit 0
```

Esempio 16-3. Redirigere, nello stesso script, sia lo `stdin` che lo `stdout` con exec

```
  1 #!/bin/bash
  2 # upperconv.sh
  3 # Converte in lettere maiuscole il testo del file di input specificato.
  4
  5 E_ACCESSO_FILE=70
  6 E_ERR_ARG=71
  7
  8 if [ ! -r "$1" ]     # Il file specificato ha i permessi in lettura?
  9 then
 10   echo "Non riesco a leggere il file di input!"
 11   echo "Utilizzo: $0 file-input file-output"
 12   exit $E_ACCESSO_FILE
 13 fi                   #  Esce con lo stesso errore anche
 14                      #+ quando non viene specificato il file di input ($1).
 15
 16 if [ -z "$2" ]
 17 then
 18   echo "Occorre specificare un file di output."
 19   echo "Utilizzo: $0 file-input file-output"
 20   exit $E_ERR_ARG
 21 fi
 22
 23
 24 exec 4<&0
 25 exec < $1            # Per leggere dal file di input.
 26
 27
 28 exec 7>&1
 29 exec > $2            #  Per scrivere nel file di output.
 30                      #  Nell'ipotesi che il file di output abbia i permessi
```

```
 31                     #+ di scrittura (aggiungiamo una verifica?).
 32
 33 # ------------------------------------------------------
 34     cat - | tr a-z A-Z   # Conversione in lettere maiuscole.
 35 #   ^^^^^               # Legge dallo stdin.
 36 #           ^^^^^^^^^^  # Scrive sullo stdout.
 37 # Comunque, sono stati rediretti sia lo stdin che lo stdout.
 38 # ------------------------------------------------------
 39
 40 exec 1>&7 7>&-       # Ripristina lo stdout.
 41 exec 0<&4 4<&-       # Ripristina lo stdin.
 42
 43 # Dopo il ripristino, la riga seguente viene visualizzata allo stdout
 44 #+ come ci aspettiamo.
 45 echo "Il file \"$1\" è stato scritto in \"$2\" in lettere maiuscole."
 46
 47 exit 0
```

La redirezione I/O è un modo intelligente per evitare il problema delle temute variabili inaccessibili all'interno di una subshell.

Esempio 16-4. Evitare una subshell

```
  1 #!/bin/bash
  2 # avoid-subshell.sh
  3 # Suggerito da Matthew Walker.
  4
  5 Righe=0
  6
  7 echo
  8
  9 cat miofile.txt | while read riga;
 10                  do {
 11                    echo $riga
 12                    (( Righe++ ));  #  I valori assunti da questa variabile
non
 13                                    #+ sono accessibili al di fuori del
ciclo.
 14                                    #  Problema di subshell.
 15                  }
 16                  done
 17
 18 echo "Numero di righe lette = $Righe"    # 0
 19                                          # Sbagliato!
 20
 21 echo "-----------------------"
 22
 23
 24 exec 3<> miofile.txt
 25 while read riga <&3
 26 do {
 27   echo "$riga"
 28   (( Righe++ ));                 #  I valori assunti da questa variabile
 29                                  #+ sono accessibili al di fuori del
ciclo.
 30                                  #  Niente subshell, nessun problema.
 31 }
 32 done
 33 exec 3>&-
 34
```

```
35 echo "Numero di righe lette = $Righe"    # 8
36
37 echo
38
39 exit 0
40
41 # Le righe seguenti non vengono elaborate dallo script.
42
43 $ cat miofile.txt
44
45 Riga 1.
46 Riga 2.
47 Riga 3.
48 Riga 4.
49 Riga 5.
50 Riga 6.
51 Riga 7.
52 Riga 8.
```

Note

[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). Meglio non utilizzare questo particolare descrittore di file.

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 Esempio 23-10). L'operatore <, posto alla fine del blocco, svolge questo compito.

Esempio 16-5. Ciclo *while* rediretto

```
 1 #!/bin/bash
 2
 3 if [ -z "$1" ]
 4 then
 5   Nomefile=nomi.data        #  È il file predefinito, se non ne viene
 6                             #+ specificato alcuno.
 7 else
 8   Nomefile=$1
 9 fi
10 #+ Nomefile=${1:-nomi.data}
11 #  può sostituire la verifica precedente (sostituzione di parametro).
12
13 conto=0
14
15 echo
16
17 while [ "$nome" != Smith ]  #  Perché la variabile $nome è stata usata
```

```
 18                            #+ con il quoting?
 19 do
 20   read nome                # Legge da $Nomefile invece che dallo stdin.
 21   echo $nome
 22   let "conto += 1"
 23 done <"$Nomefile"           # Redirige lo stdin nel file $Nomefile.
 24 #    ^^^^^^^^^^^^
 25
 26 echo; echo "$conto nomi letti"; echo
 27
 28 #  È da notare che, in alcuni linguaggi di scripting di shell più vecchi, il
 29 #+ ciclo rediretto viene eseguito come una subshell.
 30 #  Di conseguenza $conto restituirebbe 0, il valore di inizializzazione
prima
 31 #+ del ciclo.
 32 #  Bash e ksh evitano l'esecuzione di una subshell ogni volta che è
possibile,
 33 #+ cosicché questo script, ad esempio, funziona correttamente.
 34 #
 35 # Grazie a Heiner Steven per la precisazione.
 36
 37 exit 0
```

Esempio 16-6. Una forma alternativa di ciclo *while* rediretto

```
  1 #!/bin/bash
  2
  3 # Questa è una forma alternativa dello script precedente.
  4
  5 #  Suggerito da Heiner Steven
  6 #+ come espediente in quelle situazioni in cui un ciclo rediretto
  7 #+ viene eseguito come subshell e, quindi, le variabili all'interno del
ciclo
  8 #+ non conservano i loro valori dopo che lo stesso è terminato.
  9
 10
 11 if [ -z "$1" ]
 12 then
 13   Nomefile=nomi.data      #  È il file predefinito, se non ne viene
 14                           #+ specificato alcuno.
 15 else
 16   Nomefile=$1
 17 fi
 18
 19
 20 exec 3<&0                 # Salva lo stdin nel descrittore di file 3.
 21 exec 0<"$Nomefile"        # Redirige lo standard input.
 22
 23 conto=0
 24 echo
 25
 26
 27 while [ "$nome" != Smith ]
 28 do
 29   read nome               # Legge dallo stdin rediretto ($Nomefile).
 30   echo $nome
 31   let "conto += 1"
 32 done                      #  Il ciclo legge dal file $Nomefile.
 33                           #+ a seguito dell'istruzione alla riga 21.
 34
 35 #  La versione originaria di questo script terminava il ciclo "while" con
```

```
36 #+      done <"$Nomefile"
37 #  Esercizio:
38 #  Perchè questo non è più necessario?
39
40
41 exec 0<&3                   # Ripristina il precedente stdin.
42 exec 3<&-                   # Chiude il temporaneo df 3.
43
44 echo; echo "$conto nomi letti"; echo
45
46 exit 0
```

Esempio 16-7. Ciclo *until* rediretto

```
 1 #!/bin/bash
 2 # Uguale all'esempio precedente, ma con il ciclo "until".
 3
 4 if [ -z "$1" ]
 5 then
 6   Nomefile=nomi.data          #  È il file predefinito, se non ne viene
 7                               #+ specificato alcuno.
 8 else
 9   Nomefile=$1
10 fi
11
12 # while [ "$nome" != Smith ]
13 until [ "$nome" = Smith ]     # Il != è cambiato in =.
14 do
15   read nome                  # Legge da $Nomefile, invece che dallo stdin.
16   echo $nome
17 done <"$Nomefile"            # Redirige lo stdin nel file $Nomefile.
18 #    ^^^^^^^^^^^^
19
20 # Stessi risultati del ciclo "while" dell'esempio precedente.
21
22 exit 0
```

Esempio 16-8. Ciclo *for* rediretto

```
 1 #!/bin/bash
 2
 3 if [ -z "$1" ]
 4 then
 5   Nomefile=nomi.data           #  È il file predefinito, se non ne viene
 6                                #+ specificato alcuno.
 7 else
 8   Nomefile=$1
 9 fi
10
11 conta_righe=`wc $Nomefile | awk '{ print $1 }'`
12 #           Numero di righe del file indicato.
13 #
14 #  Elaborato e con diversi espedienti, ciò nonostante mostra che
15 #+ è possibile redirigere lo stdin in un ciclo "for" ...
16 #+ se siete abbastanza abili.
17 #
18 # Più conciso     conta_righe=$(wc -l < "$Nomefile")
19
20
21 for nome in `seq $conta_righe` #  Ricordo che "seq" genera una sequenza
```

```
 22                                #+ di numeri.
 23 # while [ "$nome" != Smith ]   --  più complicato di un ciclo "while"  --
 24 do
 25   read nome                    #  Legge da $Nomefile, invece che dallo
stdin.
 26   echo $nome
 27   if [ "$nome" = Smith ]       #  Sono necessarie tutte queste istruzioni
 28                                #+ aggiuntive.
 29   then
 30       break
 31   fi
 32 done <"$Nomefile"              #  Redirige lo stdin nel file $Nomefile.
 33 #    ^^^^^^^^^^^^
 34
 35 exit 0
```

Il precedente esempio può essere modificato per redirigere anche l'output del ciclo.

Esempio 16-9. Ciclo *for* rediretto (rediretti sia lo `stdin` che lo `stdout`)

```
  1 #!/bin/bash
  2
  3 if [ -z "$1" ]
  4 then
  5   Nomefile=nomi.data             #  È il file predefinito, se non ne viene
  6                                  #+ specificato alcuno.
  7 else
  8   Nomefile=$1
  9 fi
 10
 11 Filereg=$Nomefile.nuovo          #  Nome del file in cui vengono salvati i
 12                                  #+ risultati.
 13 NomeFinale=Jonah                 #  Nome per terminare la "lettura".
 14
 15 conta_righe=`wc $Nomefile | awk '{ print $1 }'` #  Numero di righe del file
 16                                                 #+ indicato.
 17
 18 for nome in `seq $conta_righe`
 19 do
 20   read nome
 21   echo "$nome"
 22   if [ "$nome" = "$NomeFinale" ]
 23   then
 24       break
 25   fi
 26 done < "$Nomefile" > "$Filereg"  # Redirige lo stdin nel file $Nomefile,
 27 #    ^^^^^^^^^^^^^^^^^^^^^^^^^^    e lo salva nel file di backup $Filereg.
 28
 29 exit 0
```

Esempio 16-10. Costrutto *if/then* rediretto

```
  1 #!/bin/bash
  2
  3 if [ -z "$1" ]
  4 then
  5   Nomefile=nomi.data    #  È il file predefinito, se non ne viene
  6                         #+ specificato alcuno.
  7 else
```

```
 8   Nomefile=$1
 9 fi
10
11 TRUE=1
12
13 if [ "$TRUE" ]          # vanno bene anche  if true   e   if :
14 then
15  read nome
16  echo $nome
17 fi <"$Nomefile"
18 #  ^^^^^^^^^^^^
19
20 #  Legge solo la prima riga del file.
21 #  Il costrutto "if/then" non possiede alcuna modalità di iterazione
22 #+ se non inserendolo in un ciclo.
23
24 exit 0
```

Esempio 16-11. File dati "nomi.data" usato negli esempi precedenti

```
 1 Aristotile
 2 Belisario
 3 Capablanca
 4 Eulero
 5 Goethe
 6 Hamurabi
 7 Jonah
 8 Laplace
 9 Maroczy
10 Purcell
11 Schmidt
12 Semmelweiss
13 Smith
14 Turing
15 Venn
16 Wilson
17 Znosko-Borowski
18
19 #  Questo è il file dati per
20 #+ "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 Esempio 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 Esempio 11-6). Questo permette di generare dei rapporti e dei file di log.

Esempio 16-12. Eventi da registrare in un file di log

```
 1 #!/bin/bash
 2 # logevents.sh, di Stephane Chazelas.
 3
```

```
 4 # Evento da registrare in un file.
 5 # Deve essere eseguito da root (per l'accesso in scrittura a /var/log).
 6
 7 UID_ROOT=0     # Solo gli utenti con $UID 0 hanno i privilegi di root.
 8 E_NONROOT=67   # Errore di uscita non root.
 9
10
11 if [ "$UID" -ne "$UID_ROOT" ]
12 then
13   echo "Bisogna essere root per eseguire lo script."
14   exit $E_NONROOT
15 fi
16
17
18 DF_DEBUG1=3
19 DF_DEBUG2=4
20 DF_DEBUG3=5
21
22 # Decommentate una delle due righe seguenti per attivare lo script.
23 # LOG_EVENTI=1
24 # LOG_VAR=1
25
26
27 log()  # Scrive la data e l'ora nel file di log.
28 {
29 echo "$(date)  $*" >&7     # *Accoda* la data e l'ora nel file.
30                            # Vedi oltre.
31 }
32
33
34
35 case $LIVELLO_LOG in
36  1) exec 3>&2         4> /dev/null 5> /dev/null;;
37  2) exec 3>&2         4>&2         5> /dev/null;;
38  3) exec 3>&2         4>&2         5>&2;;
39  *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
40 esac
41
42 DF_LOGVAR=6
43 if [[ $LOG_VAR ]]
44 then exec 6>> /var/log/vars.log
45 else exec 6> /dev/null               # Sopprime l'output.
46 fi
47
48 DF_LOGEVENTI=7
49 if [[ $LOG_EVENTI ]]
50 then
51   # then exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
52   # La riga precedente non funziona nella versione Bash 2.04.
53   exec 7>> /var/log/event.log        # Accoda in "event.log".
54   log                                # Scrive la data e l'ora.
55 else exec 7> /dev/null               # Sopprime l'output.
56 fi
57
58 echo "DEBUG3: inizio" >&${DF_DEBUG3}
59
60 ls -l >&5 2>&4                      # comando1 >&5 2>&4
61
62 echo "Fatto"                        # comando2
63
64 echo "invio mail" >&${DF_LOGEVENTI}  # Scrive "invio mail" nel df nr.7.
65
```

```
66
67 exit 0
```

Capitolo 17. Here document

Un *here document* è un blocco di codice con una funzione specifica. Utilizza una particolare forma di redirezione I/O per fornire un elenco di comandi a un programma o comando interattivi, come ftp, telnet, o **ex**.

```
 1 COMANDO <<InputArrivaDaQui(HERE)
 2 ...
 3 InputArrivaDaQui(HERE)
```

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

```
 1 comando nr.1
 2 comando nr.2
 3 ...
```

L'alternativa rappresentata da un "here document" è la seguente:

```
 1 #!/bin/bash
 2 programma-interattivo <<StringaLimite
 3 comando nr.1
 4 comando nr.2
 5 ...
 6 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 presenti nel here document e causare confusione.

Si noti che gli *here document* possono, talvolta, essere usati efficacemente anche con utility e comandi non interattivi.

Esempio 17-1. File di prova: crea un file di prova di due righe

```
 1 #!/bin/bash
 2
 3 # Uso non interattivo di 'vi' per scrivere un file.
 4 # Simula 'sed'.
 5
 6 E_ERR_ARG=65
 7
 8 if [ -z "$1" ]
 9 then
10   echo "Utilizzo: `basename $0` nomefile"
11   exit $E_ERR_ARG
12 fi
```

```
13
14 FILE=$1
15
16 # Inserisce 2 righe nel file, quindi lo salva.
17 #--------Inizio here document----------------#
18 vi $FILE <<x23StringaLimitex23
19 i
20 Questa è la riga 1 del file d'esempio.
21 Questa è la riga 2 del file d'esempio.
22 ^[
23 ZZ
24 x23StringaLimitex23
25 #----------Fine here document----------------#
26
27 #  Notate che i caratteri ^[ corrispondono alla
28 #+ digitazione di Control-V <Esc>.
29
30 #  Bram Moolenaar fa rilevare che questo potrebbe non funzionare con 'vim',
31 #+ a causa di possibili problemi di interazione con il terminale.
32
33 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*.

Esempio 17-2. Trasmissione: Invia un messaggio a tutti gli utenti connessi

```
 1 #!/bin/bash
 2
 3 wall <<zzz23FineMessaggiozzz23
 4 Inviate per e-mail gli ordini per la pizza di mezzogiorno
 5 all'amministratore di  sistema.
 6     (Aggiungete un euro extra se la volete con acciughe o funghi.)
 7 # Il testo aggiuntivo del messaggio va inserito qui.
 8 # Nota: 'wall' visualizza le righe di commento.
 9 zzz23FIneMessaggiozzz23
10
11 # Si sarebbe potuto fare in modo più efficiente con
12 #         wall <file-messaggio
13 #  Comunque, inesrire un messaggio campione in uno script fa risparmiare
14 #+ del lavoro.
15
16 exit 0
```

Esempio 17-3. Messaggio di più righe usando cat

```
 1 #!/bin/bash
 2
 3 #  'echo' è ottimo per visualizzare messaggi di una sola riga,
 4 #+ ma diventa problematico per messaggi più lunghi.
 5 #  Un here document 'cat' supera questa limitazione.
 6
 7 cat <<Fine-messaggio
 8 -------------------------------------
 9 Questa è la riga 1 del messaggio.
10 Questa è la riga 2 del messaggio.
11 Questa è la riga 3 del messaggio.
12 Questa è la riga 4 del messaggio.
```

```
 13 Questa è l'ultima riga del messaggio.
 14 ------------------------------------
 15 Fine-messaggio
 16
 17 #  Sostituendo la precedente riga 7 con
 18 #+   cat > $Nuovofile <<Fine-messaggio
 19 #+       ^^^^^^^^^^^^
 20 #+ l'output viene scritto nel file $Nuovofile invece che allo stdout.
 21
 22 exit 0
 23
 24
 25 #-----------------------------------------
 26 # Il codice che segue non viene eseguito per l'"exit 0" precedente.
 27
 28 # S.C. sottolinea che anche la forma seguente funziona.
 29 echo "------------------------------------
 30 Questa è la riga 1 del messaggio.
 31 Questa è la riga 2 del messaggio.
 32 Questa è la riga 3 del messaggio.
 33 Questa è la riga 4 del messaggio.
 34 Questa è l'ultima riga del messaggio.
 35 ------------------------------------------"
 36 #  Tuttavia, il testo non dovrebbe contenere doppi apici privi
 37 #+ del carattere di escape.
```

L'opzione - alla stringa limite del here document (**<<-StringaLimite**) sopprime i caratteri di tabulazione iniziali (ma non gli spazi) nell'output. Può essere utile per rendere lo script più leggibile.

Esempio 17-4. Messaggio di più righe con cancellazione dei caratteri di tabulazione

```
  1 #!/bin/bash
  2 # Uguale all'esempio precedente, ma...
  3
  4 #  L'opzione - al here document <<-
  5 #+ sopprime le tabulazioni iniziali nel corpo del documento,
  6 #+ ma *non* gli spazi.
  7
  8 cat <<-FINEMESSAGGIO
  9         Questa è la riga 1 del messaggio.
 10         Questa è la riga 2 del messaggio.
 11         Questa è la riga 3 del messaggio.
 12         Questa è la riga 4 del messaggio.
 13         Questa è l'ultima riga del messaggio.
 14 FINEMESSAGGIO
 15 # L'output dello script viene spostato a sinistra.
 16 # Le tabulazioni iniziali di ogni riga non vengono mostrate.
 17
 18 #  Le precedenti 5 righe del "messaggio" sono precedute da
 19 #+ tabulazioni, non da spazi.
 20 #  Gli spazi non sono interessati da  <<-.
 21
 22 #  Notate che quest'opzione non ha alcun effetto sulle tabulazioni
*incorporate*
 23
 24 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.

Esempio 17-5. Here document con sostituzione di parametro

```
 1 #!/bin/bash
 2 # Un altro here document 'cat' che usa la sostituzione di parametro.
 3
 4 # Provatelo senza nessun parametro da riga di comando,   ./nomescript
 5 # Provatelo con un parametro da riga di comando,   ./nomescript Mortimer
 6 # Provatelo con un parametro di due parole racchiuse tra doppi apici,
 7 #                          ./nomescript "Mortimer Jones"
 8
 9 LINEACMDPARAM=1    # Si aspetta almeno un parametro da linea di comando.
10
11 if [ $# -ge $LINEACMDPARAM ]
12 then
13   NOME=$1          # Se vi è più di un parametro,
14                    # tiene conto solo del primo.
15 else
16   NOME="John Doe"  # È il nome predefinito, se non si passa alcun parametro.
17 fi
18
19 RISPONDENTE="l'autore di questo bello script"
20
21
22 cat <<Finemessaggio
23
24 Ciao, sono $NOME.
25 Salute a te $NOME, $RISPONDENTE.
26
27 # Questo commento viene visualizzato nell'output (perché?).
28
29 Finemessaggio
30
31 # Notate che vengono visualizzate nell'output anche le righe vuote.
32 # Così si fa un "commento".
33
34 exit 0
```

Quello che segue è un utile script contenente un here document con sostituzione di parametro.

Esempio 17-6. Caricare due file nella directory incoming di "Sunsite"

```
 1 #!/bin/bash
 2 # upload.sh
 3
 4 #  Carica due file (Nomefile.lsm, Nomefile.tar.gz)
 5 #+ nella directory incoming di Sunsite/UNC (ibiblio.org).
 6 #  Nomefile.tar.gz è l'archivio vero e proprio.
 7 #  Nomefile.lsm è il file di descrizione.
 8 #  Sunsite richiede il file "lsm", altrimenti l'upload viene rifiutato.
 9
10
11 E_ERR_ARG=65
12
13 if [ -z "$1" ]
14 then
15   echo "Utilizzo: `basename $0` nomefile-da-caricare"
```

```
 16   exit $E_ERR_ARG
 17 fi
 18
 19
 20 Nomefile=`basename $1`              # Toglie il percorso dal nome del file.
 21
 22 Server="ibiblio.org"
 23 Directory="/incoming/Linux"
 24 #  Questi dati non dovrebbero essere codificati nello script,
 25 #+ ma si dovrebbe avere la possibilità di cambiarli fornendoli come
 26 #+ argomenti da riga di comando.
 27
 28 Password="vostro.indirizzo.e-mail"  # Sostituitelo con quello appropriato.
 29
 30 ftp -n $Server <<Fine-Sessione
 31 # l'opzione -n disabilita l'auto-logon
 32
 33 user anonymous "$Password"
 34 binary
 35 bell                                #  Emette un 'segnale acustico' dopo
ogni
 36                                     #+ trasferimento di file
 37 cd $Directory
 38 put "$Nomefile.lsm"
 39 put "$Nomefile.tar.gz"
 40 bye
 41 Fine-Sessione
 42
 43 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.

Esempio 17-7. Sostituzione di parametro disabilitata

```
  1 #!/bin/bash
  2 #  Un here document 'cat' con la sostituzione di parametro disabilitata.
  3
  4 NOME="John Doe"
  5 RISPONDENTE="L'autore dello script"
  6
  7 cat <<'Finemessaggio'
  8
  9 Ciao, sono $NOME.
 10 Salute a te $NOME, $RISPONDENTE.
 11
 12 Finemessaggio
 13
 14 #  Non c'è sostituzione di parametro quando si usa il quoting o l'escaping
 15 #+ sulla "stringa limite".
 16 #  Le seguenti notazioni avrebbero avuto, entrambe, lo stesso effetto.
 17 #  cat <"Finemessaggio"
 18 #  cat <\Finemessaggio
 19
 20 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.

Esempio 17-8. Uno script che genera un altro script

```
 1 #!/bin/bash
 2 # generate-script.sh
 3 # Basato su un'idea di Albert Reiner.
 4
 5 OUTFILE=generato.sh          # Nome del file da generare.
 6
 7
 8 # ------------------------------------------------------------
 9 # 'Here document contenente il corpo dello script generato.
10 (
11 cat <<'EOF'
12 #!/bin/bash
13
14 echo "Questo è uno script di shell generato (da un altro script)."
15 #  È da notare che, dal momento che ci troviamo all'interno di una subshell,
16 #+ non possiamo accedere alle variabili dello script "esterno".
17 #
18 echo "Il file prodotto si chiamerà: $OUTFILE"
19 #  La riga precedente non funziona come ci si può: normalmente attendere
20 #+ perchè è stata disabilitata l'espansione di parametro.
21 #  Come risultato avremo, invece, una visualizzazione letterale.
22
23 a=7
24 b=3
25
26 let "c = $a * $b"
27 echo "c = $c"
28
29 exit 0
30 EOF
31 ) > $OUTFILE
32 # ------------------------------------------------------------
33
34 #  Il quoting della 'stringa limite' evita l'espansione di variabile
35 #+ all'interno del corpo del precedente 'here document.'
36 #  Questo consente di conservare le stringhe letterali nel file prodotto.
37
38 if [ -f "$OUTFILE" ]
39 then
40   chmod 755 $OUTFILE
41   # Rende eseguibile il file generato.
42 else
43   echo "Problema nella creazione del file: \"$OUTFILE\""
44 fi
45
46 #  Questo metodo può anche essere usato per generare
47 #+ programmi C, Perl, Python, Makefiles e simili.
48
49 exit 0
```

È possibile impostare una variabile all'output di un here document.

```
 1 variabile=$(cat <<IMPVAR
 2 Questa variabile
 3 si estende su più righe.
 4 IMPVAR)
 5
 6 echo "$variabile"
```

Un here document può fornire l'input ad una funzione del medesimo script.

Esempio 17-9. Here document e funzioni

```
 1 #!/bin/bash
 2 # here-function.sh
 3
 4 AcquisisceDatiPersonali ()
 5 {
 6   read nome
 7   read cognome
 8   read indirizzo
 9   read città
10   read cap
11   read nazione
12 } # Può certamente apparire come una funzione interattiva, ma...
13
14
15 # Forniamo l'input alla precedente funzione.
16 AcquisisceDatiPersonali <<RECORD001
17 Ferdinando
18 Rossi
19 Via XX Settembre, 69
20 Milano
21 20100
22 ITALIA
23 RECORD001
24
25
26 echo
27 echo "$nome $cognome"
28 echo "$indirizzo"
29 echo "$città,  $cap, $nazione"
30 echo
31
32
33 exit 0
```

È possibile usare i : come comando fittizio per ottenere l'output di un here document. Si crea, così, un here document "anonimo".

Esempio 17-10. Here document "anonimo"

```
 1 #!/bin/bash
 2
 3 : <<VERIFICAVARIABILI
 4 ${HOSTNAME?}${USER?}${MAIL?}  #  Visualizza un messaggio d'errore se una
 5                               #+ delle variabili non è impostata.
 6 VERIFICAVARIABILI
 7
 8 exit 0
```

Una variazione della precedente tecnica consente di "commentare" blocchi di codice.

Esempio 17-11. Commentare un blocco di codice

```
 1 #!/bin/bash
 2 # commentblock.sh
 3
```

```
 4 : << BLOCCOCOMMENTO
 5 echo "Questa riga non viene visualizzata."
 6 Questa è una riga di commento senza il carattere "#"
 7 Questa è un'altra riga di commento senza il carattere "#"
 8
 9 &*@!!++=
10 La riga precedente non causa alcun messaggio d'errore,
11 perché l'interprete Bash la ignora.
12 BLOCCOCOMMENTO
13
14 echo "Il valore di uscita del precedente \"BLOCCOCOMMENTO\" è $?."   # 0
15 # Non viene visualizzato alcun errore.
16
17
18 #  La tecnica appena mostrata diventa utile anche per commentare
19 #+ un blocco di codice a scopo di debugging.
20 #  Questo evita di dover mettere il "#" all'inizio di ogni riga,
21 #+ e quindi di dovere, più tardi, ricominciare da capo e cancellare tutti i
"#"
22
23 : << DEBUGXXX
24 for file in *
25 do
26  cat "$file"
27 done
28 DEBUGXXX
29
30 exit 0
```

Un'altra variazione di questo efficace espediente rende possibile l'"auto-documentazione" degli script.

Esempio 17-12. Uno script che si auto-documenta

```
 1 #!/bin/bash
 2 # self-document.sh: script autoesplicativo
 3 # È una modifica di "colm.sh".
 4
 5 RICHIESTA_DOCUMENTAZIONE=70
 6
 7 if [ "$1" = "-h"  -o "$1" = "--help" ]     # Richiesta d'aiuto.
 8 then
 9   echo; echo "Utilizzo: $0 [nome-directory]"; echo
10   sed --silent -e '/DOCUMENTAZIONEXX$/,/^DOCUMENTAZIONEXX$/p' "$0" |
11   sed -e '/DOCUMENTAZIONEXX$/d'; exit $RICHIESTA_DOCUMENTAZIONE; fi
12
13
14 : << DOCUMENTAZIONEXX
15 Elenca le statistiche di una directory specificata in formato tabellare.
16 ---------------------------------------------------------------------------
--
17 Il parametro da riga di comando specifica la directory  di cui si desiderano
18 le statistiche. Se non è specificata alcuna directory o quella indicata non
19 può essere letta, allora vengono visualizzate le statistiche della directory
20 di lavoro corrente.
21
22 DOCUMENTAZIONEXX
23
24 if [ -z "$1" -o ! -r "$1" ]
25 then
26   directory=.
```

```
27 else
28   directory="$1"
29 fi
30
31 echo "Statistiche di "$directory":"; echo
32 (printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
33 ; ls -l "$directory" | sed 1d) | column -t
34
35 exit 0
```

Vedi anche Esempio A-26 come eccellente dimostrazione di script autoesplicativo.


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)
```


Alcune utility non funzionano se inserite in un *here document*.


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*. Allo stesso modo uno spazio posto dopo la stringa limite provoca comportamenti imprevisti. Lo spazio impedisce il riconoscimento della stringa limite.

```
  1 #!/bin/bash
  2
  3 echo "------------------------------------------------------------------
------"
  4
  5 cat <<StringaLimite
  6 echo "Questa è la riga 1 del messaggio contenuto nel here document."
  7 echo "Questa è la riga 2 del messaggio contenuto nel here document."
  8 echo "Questa è la riga finale del messaggio contenuto nel here
document."
  9      StringaLimite
 10 #^^^^String limite indentata. Errore! Lo script non si comporta come
speravamo.
 11
 12 echo "------------------------------------------------------------------
------"
 13
 14 #  Questi commenti si trovano esternamente al 'here document'
 15 #+ e non vengono visualizzati.
 16
 17 echo "Fuori dal here document."
 18
 19 exit 0
 20
 21 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.

17.1. Here String

Una *here string* può essere considerata un *here document* ridotto ai minimi termini. È formata semplicemente da **COMANDO <<<$PAROLA**, dove `$PAROLA` viene espansa per diventare lo `stdin` di `COMANDO`.

Esempio 17-13. Anteporre una riga in un file

```
 1 #!/bin/bash
 2 # prepend.sh: Aggiunge del testo all'inizio di un file.
 3 #
 4 #  Esempio fornito da Kenny Stauffer,
 5 #  leggermente modificato dall'autore del libro.
 6
 7
 8 E_FILE_ERRATO=65
 9
10 read -p "File: " file   # L'opzione -p di 'read' visualizza il prompt.
11 if [ ! -e "$file" ]
12 then   # Termina l'esecuzione se il file non esiste.
13   echo "File $file non trovato."
14   exit $E_FILE_ERRATO
15 fi
16
17 read -p "Titolo: " titolo
18 cat - $file <<<$titolo > $file.nuovo
19
20 echo "Il file modificato è $file.nuovo"
21
22 exit 0
23
24 # da "man bash"
25 # Here Strings
26 #       A variant of here documents, the format is:
27 #
28 #              <<<word
29 #
30 #       The word is expanded and supplied to the command on its standard
input.
```

Esercizio: scoprite altri impieghi per le *here string*.