Introduzione allo Shell Scripting

Domenico Delle Side

A papà

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questo primo capitolo intende chiarire alcune convenzioni usate in questo documento ed allo stesso tempo fornire al lettore dei concetti introduttivi sull'uso della documentazione disponibile in ambiente *Unix*.

1.1 Perché questo documento

Chiunque abbia usato in maniera non passiva un calcolatore si sarà chiesto come questo riesca a distinguere le istruzioni che gli vengono impartite. Questo processo consiste nel prendere un input dall'utente, processarlo e darlo al calcolatore. In ambienti *Unix/POSIX* queste funzioni vengono svolte da un apposito software, la *Shell* o interprete di comandi .

Sfruttando le caratteristiche degli interpreti di comandi, è possibile realizzare semplici script in grado di automatizzare processi lunghi e noiosi. Si pensi a quanto fastidioso sarebbe dover convertire i nomi di 200 file tutti in lettere minuscole. Un lavoro improbo solo a pensarci. Grazie a poche righe di codice, però, questo problema può diminuire enormente la sua difficoltà ed essere svolto in pochi istanti. Un altro uso comune degli script di shell è quello di essere usati come banco di prova per progetti di software. Prima di realizzare un programma, si può cercare di farne un clone usando uno script, in modo da verificarne usabilità e funzionalità.

Questo testo si propone di fornire una semplice e concisa introduzione allo shell scripting, puntando maggiormente al lato pratico della questione. Allo stesso tempo, si cerca di colmare quella che per molti utenti italiani è una pecca del mondo *GNU/Linux*. Se escludiamo il magnifico lavoro svolto dal *PLUTO* e da *Daniele Giacomini* con i suoi *Appunti di Informatica Libera*, il panorama della documentazione in italiano è piuttosto desolato.

Il materiale è corredato da numerosi esempi; è auspicabile che il lettore li provi sulla propria macchina, cercando allo stesso tempo di personalizzarli. Si è scelto di non trattare singolarmete i comandi disponibili in un sistema *Unix*, questi verranno introdotti brevemente negli esempi, dandone una minima spiegazione. Il lettore interessato potrà consultare autonomamente la pagina manuale del comando in esame per approfondire le proprie conoscenze.

Per il momento, il lavoro intende essere un testo introduttivo, incompleto sotto molti aspetti; nonostante ciò, si spera che con il tempo e con le segnalazioni dei lettori possa crescere e diventare un punto di riferimento per ogni utente che desidera avvicinarsi all'argomento. Suggerimenti e critiche saranno sempre bene accetti, così come le considerazioni circa l'organizzazione dei contenuti. Per comunicare queste impressioni, scrivete a Domenico Delle Side (`<nicodds@Tiscali.IT>`).

FIXME: Scrivere qualcosa di più bello, questa introduzione non mi piace.

1.2 Requisiti minimi

Per leggere in maniera proficua il documento è necessario avere una minima familiarità con la linea di comando. Allo stesso tempo, occorrerebbe anche conoscere i vari comandi messi a disposizione dai sistemi *Unix*. Sono necessarie inoltre una buona dose di pazienza e di buona volontà.

Dal lato tecnico è anche necessario avere un buon editor di testi; il mio consiglio è di usare l'onnipotente *Emacs* , che ha tutto ciò di cui un programmatore possa avere bisogno. Un altro ottimo editor è *vi* , o la sua versione migliorata *vim*.

Un altro elemento essenziale è ovviamente avere una shell *Bash*. Alcuni degli argomenti trattati, infatti, sono caratteristici di questo particolare interprete di comandi e potrebbero non essere transponibili ad altre shell.

1.3 Convenzioni

Nel corso della discussione verranno usati caratteri differenti per sottolineare alcuni concetti. In particolare si userà il *corsivo* per i nomi di persone e di cose (e.g. *Bash*, *Unix*, *Mario Rossi*, . . .); il **grassetto** per dare maggior peso ad una parola o ad un concetto in una frase.

Gli esempi di codice ed i comandi *Unix* saranno tutti scritti con un carattere tipo `macchina da scrivere` (e.g. `#!/bin/bash`, `echo`, . . .). Ogni esempio sarà preceduto da un titoletto del tipo "**Esempio x.y.z**"[1] che lo numererà e seguito immediatamente da una sezione "**Come funziona:**" che ne chiarirà il funzionamento.

1.4 Pagine manuale

Una delle caratteristiche più affascinanti del mondo *Unix* è la quantità smisurata di informazioni che ne accompagnano il software. Ogni comando infatti possiede la sua **pagina manuale** (manpage) in cui vengono descritti brevemente il suo funzionamento, i parametri accettati, ecc . . .

[1] Spieghiamo la convenzione utilizzata: x rappresenta il numero del capitolo, y il numero della sezione e z il numero progressivo dell'esempio.

Accedere alla pagina manuale di un comando è piuttosto semplice, è sufficiente infatti digitare in un terminale il comando `man nome_comando`. Le pagine manuale sono divise in sezioni, ognuna riguardante un argomento specifico, in particolare:

- Sezione 1: Comandi accessibili all'utente, applicazioni
- Sezione 2: Chiamate di sistema e codici d'errore del kernel
- Sezione 3: Funzioni di libreria
- Sezione 4: Device driver e protocolli di rete
- Sezione 5: Formati standard dei file
- Sezione 6: Giochi
- Sezione 7: File e documenti misti
- Sezione 8: Comandi d'amministrazione
- Sezione 9: Specifiche ed interfacce del kernel nascoste

Qualora il formato elettronico delle manpage fosse scomodo, si può facilmente stampare la pagina in questione con il l'istruzione `man -t nome_comando | lpr`, o `man -t nome_comando > nome_comando.ps` se si prefersice il formato postscript.

Se si ricerca la pagina manuale senza conoscere il nome esatto di un programma, si può tentare con i comandi `apropos` e `whatis`. Il primo cerca nel loro database comune per una specifica stringa, mentre il secondo per una parola.
Ad esempio:

```
[nico@deepcool nico]$ whatis whatis
whatis          (1)  - search the whatis database for complete words
```

1.5 Licenza

Il documento è rilasciato al pubblico sotto licenza GNU FDL (GNU *Free Documentation License*). Per i termini della licenza, si consulti l'appendice A.

Anche se molti di questi sono banali, tutti gli esempi sono rilasciati sotto licenza GNU GPL (GNU *General Public License*). Una copia della licenza può essere reperita all'indirizzo `http://it.gnu.org/licenses/gpl.html`.

Chi volesse avere maggiori informazioni sul progetto GNU e sui suoi successi può consultare direttamente il suo sito Internet presso `http://www.gnu.org`, una buona parte del quale viene tradotto e tenuto aggiornato in lingua italiana dal *Gruppo dei Traduttori Italiani dei Testi del Progetto GNU* (`<http://www.softwarelibero.it/gnudoc/>`).

1.6 Ringraziamenti

Un grazie a tutti coloro che mi hanno aiutato nella stesura di questo lavoro.

- Aggiungere
- i
- nomi

Capitolo 2

Sintassi, variabili ed operatori

In questo capitolo si daranno le basi sintattiche della shell *Bash*, dando ampio spazio alla trattazione delle *variabili* e dei *parametri*. Verranno anche presi in considerazione argomenti quali gli operatori più importanti della shell *Bash* e le condizioni.

2.1 Sintassi di base

Uno dei modi migliori di introdurre gli elementi sintattici degli script di shell è partire dai commenti. In *Bash*, come in altre shell *Unix*, i commenti vengono individuati dal carattere #. Per la precisione, tutto ciò che segue ed è sulla stessa riga di # è da considerarsi un commento. Ad esempio le linee seguenti rappresentano commenti:

```
# Questo è un commento che si estende su una linea
#
# Questo è un commento
# piazzato su due linee
#
funzione() # questo è un commento piazzato dopo una funzione
#
```

Una caratteristica essenziale di uno shell script è quella di dover sempre iniziare con un uno strano simbolo: `#!` (che non è un commento!). Questa combinazione di caratteri viene chiamata *Shee-Bang* e serve a specificare quale deve essere l'interprete delle istruzioni che seguono. Nel nostro caso, il primo rigo di ogni script sarà:

```
#!/bin/bash
# Shee-Bang seguito dal percorso completo dell'interprete
#
# Seguono le istruzioni dello script

...
```

2.1.1 Parole speciali

Prima di procedere oltre, chiariamo che con "parola" si intende una qualsiasi sequenza di uno o più caratteri che *Bash* interpreta come una singola entità. Ciò detto, Osserviamo che *Bash* riserva alcune parole (Per la maggior parte comandi) per attribuir loro un significato speciale. Queste sono:

```
!  case do done elif else fi for function if in
     select then until while { } time [[ ]]
```

Queste parole possono essere usate al di fuori del loro significato speciale, ma con le dovute accortezze. Ad esempio, è perfettamente legale[1] utilizzare una delle parole riservate come argomento di un comando:

```
[nico@deepcool nico]$ echo if
if
[nico@deepcool nico]$ echo function
function
```

Un altro modo per utilizzare le parole elencate consiste nel racchiuderle tra virgolette (Anche apici, si tratta comunque di quoting) o nel farle precedere dal carattere "\"(Escaping). In questo modo, la shell non andrà incontro ad alcuna ambiguità nell'interpretare le istruzioni.

I tempi sono ormai sufficientemente maturi per cimentarci in un primo script, proviamo con il classico "Ciao mondo":

Esempio 2.1.1

```
#!/bin/bash
#
# Ciao mondo con la shell Bash

EXIT_SUCCESS=0

echo -e "Ciao mondo\n"

exit $EXIT_SUCCESS
```

Come funziona:

Lo script[2] inizia, come necessario, con `#!/bin/bash`, seguito da due commenti. Successivamente viene definita la variabile `EXIT_SUCCESS` assegnandole il valore 0. La riga seguente fa uso di un comando presente all'interno della shell; `echo` serve a stampare sullo schermo del testo (per ragioni di portabilità `echo` è fornito anche come comando esterno, tramite il pacchetto *GNU Shell Utils*). È stato usato con l'opzione `-e` che consente di interpretare correttamente la sequenza di escape `\n` , che rappresenta un ritorno a capo. Per maggiori informazioni, si digiti in un terminale `man echo`. Lo script termina con la riga `exit $EXIT_SUCCESS`. Anche `exit` è un comando interno alla shell (*Bash built-in command*) e serve in un certo senso a stabilire una comunicazione tra lo script e l'ambiente in cui viene eseguito. Una volta terminato, infatti,

[1] Legale secondo la sintassi di *Bash*, non allarmatevi!

[2] Per eseguire con successo lo script è necessario fornirlo dei giusti permessi. chmod 755 nome_script darà all'utente il permesso di esecuzione sullo script

il nostro script dirà alla shell che tutto è andato per il verso giusto (codice 0). È convenzione che il valore di ritorno di uno script, o qualsiasi altro programma, sia 0 in caso di riuscita, un qualsiasi altro numero in caso d'errore[3]. Ad esempio, se un comando non viene trovato, il valore di ritorno del processo è 127; se il comando viene trovato, ma non è eseguibile, il valore di ritorno è 126.

In questo primo script abbiamo incontrato alcune novità, una di queste è rappresentata dalla definizione di una variabile. Questo argomento sarà oggetto della prossima sezione.

2.2 Variabili

Spesso capita di cucinare e di conservare le pietanze in contenitori per poi usarle in futuro. La shell fornisce "contenitori" analoghi, le **variabili**, nelle quali è possibile memorizzare valori per usarli in futuro o per maneggiarli più agevolmente.

2.2.1 Variabili locali

Nella sezione precedente abbiamo avuto già un primo incontro con le variabili. Abbiamo imparato in particolare come dichiararne una ed assegnarle un valore:

```
EXIT_SUCCESS=0
```

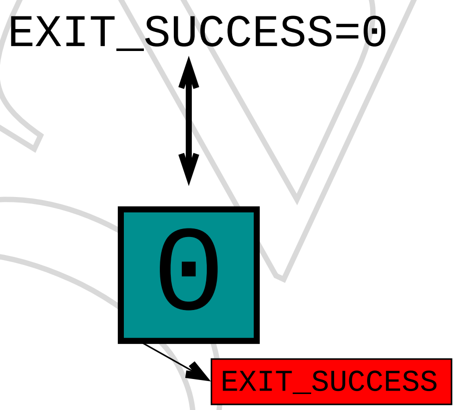

Figura 2.1: Schematizzazione dell'assegnazione di una variabile

In fase di assegnazione, occorre fare attenzione a non lasciare spazi tra il nome della variabile il segno = ed il valore, in caso contrario la shell interpreterebbe = come un operatore (cfr. pag. 13, sez. 2.4). Un altro modo di assegnare una variabile fa uso del comando `read` , un'altro comando interno della *Bash*. Ad esempio, il seguente codice legge dalla tastiera un valore e lo assegna alla variabile `$VAR`:

```
read VAR
```

Si noti che nel dichiarare una variabile occorre sempre usarne il nome privo del simbolo "$", che va invece usato quando si deve far riferimento ad essa. Chiariamo la questione con un esempio:

[3] In /usr/include/asm/errno.h è presente una lista di codici d'errore

Esempio 2.2.1

```
#!/bin/bash
#
# Dichiarazione di variabili

EXIT_SUCCESS=0
NOME=Nico

echo "Il mio nome è $NOME, il tuo?"
read TUO_NOME
echo "Ciao $TUO_NOME, buon divertimento con la shell!"

exit $EXIT_SUCCESS
```

Come funziona:

Lo script inizia con il solito `#!/bin/bash` (d'ora in poi non lo faremo più notare) e subito dopo vengono definite le variabili `$EXIT_SUCCESS` e `$NOME`[4].

L'istruzione `echo`, successivamente, stampa sullo schermo il testo `Il mio nome è Nico, il tuo?` ed attende finché non viene scritto qualcosa e poi premuto il tasto return(`RET`). L'input letto dalla tastiera con `read` viene assegnato alla variabile `$TUO_NOME` che viene usata per stampare un messaggio di cortesia. Lo script ritorna alla shell il valore 0.

2.2.2 Variabili d'ambiente

Le variabili che abbiamo incontrato fino ad ora cessano di esistere non appena lo script termina. Nonostante ciò, esistono alcune variabili, dette **d'ambiente** che conservano nel tempo il loro valore. Tali variabili sono accessibili all'utente attraverso la parola chiave `export`[5] in modo da garantirne un uso corretto. Occorre comunque notare che, anche usando `export` all'interno di uno script su di una variabile d'ambiente, i cambi a tale variabile rimarranno locali allo script e non si estenderanno alla shell che lo ha eseguito. Ciò accade perché uno script può esportare variabili solo ai suoi processi figli, cioè alle istruzioni in esso presenti. Risulta ovvio a questo punto che le variabili d'ambiente sono variabili impostate primordialmente dalla shell o altri programmi (Ad esempio `login` e `telnet`) e sono fruibili dai suoi processi figli (i comandi, gli script, ...).

Una lista di variabili globali può essere ottenuta digitando in console l'istruzione `set`. Alcune delle variabili più comuni sono[6]:

- `$HOME` - la home directory dell'utente

[4] D'ora in poi, quando si farà riferimento ad una variabile si userà sempre la scrittura $NOME_VARIABILE

[5] La keyword export è caratteristica della shell *Bash*, con la shell *(t)csh*, ad esempio, si usa la parola setenv

[6] Si tenga presente il fatto che non tutte queste variabili sono impostate dalla shell. Per ottenere una collezione di variabili d'ambiente proprie di *Bash*, si consulti [3].

- $HOSTNAME - hostname del calcolatore
- $HOSTTYPE - il tipo di hardware del calcolatore
- $IFS - la lista dei separatori di campi
- $LOGNAME - il nome utente con cui si è effettuato il login
- $OSTYPE - il nome del sistema operativo del calcolatore
- $PATH - la lista delle directory dei comandi
- $PWD - la directory di esecuzione dello script
- $SHELL - la shell di default dell'utente

Cerchiamo di saperne di più.

Esempio 2.2.2

```
#!/bin/bash
#
# Uso delle variabili d'ambiente

EXIT_SUCCESS=0

echo "Informazioni sul sistema:"
echo -e "hostname:\t$HOSTNAME"
echo -e "hardware:\t$HOSTTYPE"
echo -e "OS:\t$OSTYPE\n"

echo "Informazioni sull'utente:"
echo -e "logname:\t$LOGNAME"
echo -e "homedir:\t$HOME"
echo -e "shell:\t$SHELL"
echo -e "path:\t$PATH\n"

echo -e "Directory di esecuzione: $PWD\n"

echo "Esportiamo un nuovo PATH..."

export PATH=${PATH}:/usr/games

echo -e "...il nuovo PATH è:\n$PATH"

exit $EXIT_SUCCESS
```

Come funziona:

Lo script acquisisce alcuni dati tramite delle variabili d'ambiente stampandoli sullo schermo grazie al comando `echo`. È da notare l'uso della sequenza di escape `\t`, che corrisponde ad un `TAB` . Per far interpretare correttamente la sequenza di escape, è necessario racchiudere tra virgolette la stringa che la contiene. Le righe seguenti ci insegneranno molto. Prima di tutto possiamo notare l'uso di `export`:

```
export NOME_VARIABILE=valore_della_variabile
```

o, equivalentemente

```
NOME_VARIABILE=valore_della_variabile
export NOME_VARIABILE
```

Un'altra novità è rappresentata dall'uso della scrittura `${PATH}`. In realtà, questo è il vero modo per riferirsi ad una variabile, in genere si preferisce la più comoda forma `$PATH`, ma questa deve essere subito abbandonata quando si corre il rischio di incappare in ambiguità.

Come si può vedere dall'output dello script, il contenuto di `$PATH` sembra essere variato, ma se, finito lo script, digitiamo in terminale `echo $PATH`, possiamo renderci conto che il cambio della variabile è rimasto confinato nello "spazio" di esecuzione dello script.

Proviamo ora una versione leggermente modificata e più breve dello script precedente.

Esempio 2.2.3

```
#!/bin/bash
#
# ancora sulle variabili

EXIT_SUCCESS=0

echo "Informazioni sul sistema:"
echo -e "hostname:\t$(hostname)"
echo -e "hardware:\t$(uname -m)"

exit $EXIT_SUCCESS
```

Come funziona:

Come si può notare eseguendolo, il risultato di questo script è pressocché identico al precedente, la differenza sostanziale sta nel codice. Anziché utilizzare variabili d'ambiente, questa volta si è usato come variabile l'output del comando `hostname` prima e `uname -m` poi. Tali comandi sono stati racchiusi all'interno della combinazione di simboli `$(...)`[7] che rappresenta per la *Bash* una sorta di variabile temporanea in cui memorizzare il risultato visibile di un comando (In realtà, ciò che si realizza è un'espansione di comandi).

[7] L'uso di `$(...)` è piuttosto recente, in passato si usava racchiudere il comando da eseguire all'interno di apici rovesciati, `` `...` ``

2.3 Parametri

Un parametro è un'entità che immagazzina un valore; può essere indicato tramite un nome, un numero o un carattere speciale. Vediamo in questo modo che i parametri sono una generalizzazione delle variabili, infatti una variabile è un caso di parametro identificato da una parola.

Immaginiamo di lanciare un nostro script in maniera un po' differente, supponiamo di farlo scrivendo in un terminale:

```
[nico@deepcool intro]$ ./script param1 param2 param3
```

Come sfrutterà lo script queste informazioni aggiuntive? Ritornerà un errore?
Chiariamo il tutto con un esempio.

Esempio 2.3.1

```
#!/bin/bash
#
# Utilizzo dei parametri posizionali

EXIT_SUCCESS=0

echo "Hai inserito ${1}, $2 e $3 per un totale di $# parametri"

exit $EXIT_SUCCESS
```

Come funziona:

Lo script si sviluppa su un'unica riga; $1[8], $2 ed $3 rappresentano rispettivamente il primo, secondo e terzo parametro passato allo script, mentre $# contiene il numero di parametri passati.

Proviamo ad eseguire lo script:

```
[nico@deepcool intro]$ ./script param1 param2 param3
Hai inserito param1, param2 e param3 per un totale di 3 parametri
```

Sembra funzionare bene, ma cosa succederebbe se inserissimo quattro parametri anziché tre? Per il momento lasciamo in sospeso questa domanda, promettendo di occuparcene più in avanti nel capitolo 3.

I parametri disponibili all'interno di uno script non si esauriscono di certo qui, alcuni ulteriori sono[9]:

[8] È legale anche il parametro $0 che rappresenta il nome dello script

[9] Per avere l'elenco completo dei parametri disponibili con una breve spiegazione, si consulti la pagina manuale di bash

- `$$` - process id (`PID`) dello script
- `$?` - valore di ritorno di un comando, funzione o dello stesso script
- `$*` - stringa contenente tutti i parametri posizionali passati
- `$@` - insieme di tutti i parametri posizionali passati

È importante chiarire la differenza che corre tra `$*` e `$@`; la prima scrittura fornisce tutti i parametri posizionali visti come un'unica parola, mentre la seconda fornisce tutti i parametri posizionali ognuno come singola parola. Questa distinzione sarà importante in futuro.

2.3.1 Espansione di parametri

In precedenza, abbiamo già incontrato la forma `${VARIABILE}` (sezione 2.2.2) ed abbiamo affermato che questo è il modo più generale per riferirsi ad una variabile. Tuttavia, il simbolo `${ ... }` ha un ruolo ben più importante, poiché rappresenta la forma più generale per consentire alla shell di espandere parametri. Nel caso di `${VARIABILE}` il parametro è la variabile `$VARIABILE` e l'espansione è il suo valore.

Vediamo ora alcune delle altre forme di espansione offerte dalla shell *Bash*.

- `${PARAMETRO:-PAROLA}`
  Se `PARAMETRO` non è dichiarato o non ha valore, viene sostituita l'espansione di `PAROLA`, altrimenti viene sostituito il valore di `PARAMETRO`.
- `${PARAMETRO:+PAROLA}`
  Se `PARAMETRO` non è dichiarato o non ha valore non viene sostituito nulla, altrimenti viene sostituita l'espansione di `PAROLA`.
- `${PARAMETRO:=PAROLA}`
  Se `PARAMETRO` non è dichiarato o non ha valore, gli viene assegnata l'espansione di `PAROLA`, altrimenti `PARAMETRO` mantiene il suo valore.
- `${PARAMETRO:?PAROLA}`
  Se parametro non è dichiarato o non ha valore, viene stampata sullo standard error l'espansione di `PAROLA`, altrimenti viene sostituito il valore di `PARAMETRO`.
- `${PARAMETRO:POSIZIONE}`
  `${PARAMETRO:POSIZIONE:LUNGHEZZA}`
  Vengono sostituiti "`LUNGHEZZA`" caratteri di `PARAMETRO` a partire da `POSIZIONE`. Se `LUNGHEZZA` è omessa, viene sostituita la sottostringa di `PARAMETRO` a partire da `POSIZIONE`. `LUNGHEZZA` deve essere un numero maggiore o uguale a zero, mentre `POSIZIONE` può anche essere minore di zero, in tal caso il posizionamento viene effettuato a partire dalla fine di `PARAMETRO`.
- `${#PARAMETRO}`
  Viene sostituita la lunghezza in caratteri dell'espansione di `PARAMETRO`.

- `${PARAMETRO#PAROLA}`
  `${PARAMETRO##PAROLA}`
  Se presente un solo # elimina dall'inizio dell'espansione di `PARAMETRO` la più breve occorrenza dell'espansione di `PAROLA`, altrimenti, qualora fossero presenti due #, ne rimuove l'occorrenza pi lunga. Per capirne meglio il funzionamento, si provi a digitare in un terminale questa linea di codice:
  `stringa=abcABC123ABCabc && echo ${stringa#a*C} && echo ${stringa##a*C}`.

- `${PARAMETRO%PAROLA}`
  `${PARAMETRO%%PAROLA}`
  Se presente un solo %, rimuove a partire dalla fine dell'espansione di `PARAMETRO` la più breve occorrenza dell'espansione di `PAROLA`, altrimenti, qualora fossero presenti due %, ne toglie l'occorrenza più lunga. Ad esempio:
  `stringa=abcABC123ABCabc && echo ${stringa%C*c} && echo ${stringa%%C*c}`.

- `${PARAMETRO/PATTERN/STRINGA}`
  `${PARAMETRO//PATTERN/STRINGA}`
  Se presente un solo /, rimpiazza la prima occorrenza di `PATTERN` all'interno dell'espansione di `PARAMETRO` con `STRINGA`. Con due /, la sostituzione viene effettuata su ogni occorrenza, non solo la prima. Come esempio, si verifichi il risultato di:
  `string=abcABC123ABCabc && echo ${string/abc/xyz} && echo ${string//abc/xyz}`.

2.4 Operatori e condizioni

Una condizione è un particolare avvenimento per il quale è lecito domandarsi se sia vero o meno. Le condizioni fanno parte della vita quotidiana di ogni persona; ogni nostra azione, ad esempio, ha luogo solo in virtù di determinate circostanze. Se vediamo l'indicatore del carburante nella zona rossa, è del tutto naturale andare a fare rifornimento; ciò accade perché abbiamo valutato la condizione fornitaci dalla lancetta dell'indicatore. La shell *Bash*, pur non funzionando a benzina, fa altrettanto; valuta le condizioni che le vengono fornite ed agisce di conseguenza.

Valutare una condizione è una caratteristica essenziale per ogni linguaggio di programmazione, pertanto verranno elencati alcuni degli operatori più importanti della shell *Bash*.

2.4.1 Operatori su numeri

Divideremo gli operatori che agiscono su numeri in due classi:

- Operatori di confronto:
  quelli che eseguono un controllo circa l'ordine tra due numeri.

- Operatori aritmetici:
  quelli che manipolano i numeri per ottenerne di nuovi.

Operatori di confronto

I seguenti sono operatori binari su numeri, ovvero agiscono su 2 numeri

- `-eq`
  verifica l'uguaglianza tra due numeri (`5 -eq 4`). "`-eq`" deriva dalla lingua inglese, in cui uguarle si dice "**eq**ual".

- `-ne`
  verifica che due numeri siano differenti (`5 -ne 4`). Anche in questo caso l'origine va cercata nella lingua anglosassone, `-ne` sta per "**n**ot **e**qual".

- `-gt`
  verifica che il primo numero fornito sia maggiore del secondo (`5 -gt 4`, deriva da "**g**reather **t**han")

- `-ge`
  verifica che il primo numero sia maggiore o al più uguale al secondo (`5 -ge 4`, da "**g**reather or **e**qual")

- `-lt`
  verifica che il primo numero fornito sia minore del secondo (`5 -lt 4`, da "**l**ess **t**han")

- `-le`
  verifica che il primo numero sia minore o al più uguale al secondo (`5 -le 4`, da "**l**ess or **e**qual")

Operatori aritmetici

Oltre alle 4 operazioni, la shell *Bash* è dotata di altri operatori aritmetici, molti dei quali sono simili a quelli del *C*.

- `+`
  somma 2 numeri (`2 + 2`)

- `-`
  sottrae dal primo il secondo numero

- `*`
  moltiplica due numeri

- `/`
  divide il primo numero per il secondo

- `**`
  esponenzia il primo numero al secondo (`2 ** 3` equivale a 2^3)

- `%`
  ritorna il resto intero della divisione tra il primo ed il secondo numero

La shell consente di eseguire calcoli utilizzanto alcune sintassi particolari. Occorre, però, notare che è possibile eseguire solo operazioni su numeri interi; per questo motivo, il risultato di calcoli come `2 / 3` viene arrotondato al più grande intero minore o uguale al risultato (Parte intera). Per effettuare calcoli con numeri in virgola mobile, si può utilizzare il comodo programma `bc` .

Il calcolo di operazioni aritmetiche può essere effettuato utilizzando il comando `expr` , o il comando interno `let` , oppure racchiudendo l'espressione da valutare nel simbolo `$(( ... ))` (Espansione aritmetica). Inoltre, possono anche essere usata la forma `(( ... ))`, che non produce alcun output, ma valuta solamente l'espressione aritmetica contenuta al suo interno.

Esempio 2.4.1

```
#!/bin/bash
#
# Esecuzione di calcoli
EXIT_SUCCESS=0

echo $(( 2 / 3 ))
echo $(( 5 + 5 / 2 ))
echo $(( $(( 5 + 5 )) / 2 ))
echo $(( 2 ** 8 ))

exit $EXIT_SUCCESS
```

Come funziona:

L'esempio riportato illustra l'uso di alcuni semplici operatori aritmetici e del simbolo `$(( ... ))`. Le precedenze tra gli operatori sono uguali a quelle usate comunemente, ad esempio `5 + 5 / 2` equivale a `5 + (5 / 2)`.
L'esempio ci dice anche che è possibile annidare più simboli `$(( ... ))` nel calcolo di una espressione.

Si tenga presente che sono legali anche le forme *"alla C"* +=, -=, *=, /= e %=; queste devono essere interpretate come:

```
VARIABILE_1(OP)=$VARIABILE_2
        equivale a
VARIABILE_1=$(( $VARIABILE_1 (OP) $VARIABILE_2 ))
```

dove `(OP)` è uno tra +, -, *, / e %.

Esempio 2.4.2

```
#!/bin/bash
#
# Esecuzione di calcoli (2)
EXIT_SUCCESS=0
```

```
VARIABILE=10

echo $VARIABILE

(( VARIABILE+=10 ))

echo $VARIABILE

(( VARIABILE/=10 ))

echo $VARIABILE

exit $EXIT_SUCCESS
```

Come funziona:

L'esempio illustra l'uso delle forme `(OP)=` nonché quello di `(( ... ))`. La forma `(( ... ))`, come detto, non produce alcuna espansione, ma si limita a modificare il valore di `$VARIABILE`.

2.4.2 Operatori logici

Di seguito saranno riportati gli operatori logici forniti dalla shell *Bash* e se ne chiarirà il significato nella sezione 2.6.

- `!`
  Operatore di negazione, inverte il valore logico dell'espressione al quale viene applicato (`!(vero) = falso`).
- `-a`
  Operatore AND logico. Ritorna "vero" se entrambi gli argomenti passati sono "veri".
- `-o`
  Operatore OR logico. Ritorna vero se almeno uno degli argomenti è vero.

2.4.3 Operatori su stringhe

Riportiamo gli operatori su stringhe; la maggior parte è binaria, solo due operatori sono unari.

- `-n`
  Ritorna vero se la stringa cui è applicato è di lunghezza maggiore di zero.
- `-z`
  Ritorna vero se la stringa cui è applicato è di lunghezza zero.
- == o =
  Ritorna vero se le stringhe passate come argomenti sono uguali (`''abc'' == ''abc''`).

- !=
  Ritorna vero se le stringhe passate come argomenti sono differenti
  (`‘‘abc’’ != ‘‘cde’’`).

- <
  Ritorna vero se la prima stringa passata come argomento è lessicograficamente minore rispetto alla seconda (`‘‘abc’’ < ‘‘z’’`). Può essere usato anche in accoppiata con =, ovvero <=.

- >
  Ritorna vero se la prima stringa passata come argomento è lessicograficamente maggiore rispetto alla seconda (`‘‘z’’ > ‘‘abc’’`). Può essere usato anche in accoppiata con =, ovvero >=.

Per usare < e > occorre fare attenzione a precederli con il carattere di escape "\"o racchiuderli tra virgolette (O apici), la shell altrimenti li interpreterebbe come operatori di input/output, come vedremo nella sezione 2.4.5

2.4.4 Operatori su file

Gli operatori su file sono per la maggior parte operatori unari (accettano un solo input), tranne gli ultimi 3 di questa lista, che sono binari. Ritornano tutti un valore booleano (vero o falso) a seconda che il test effettuato abbia avuto successo o meno.

- `-a`
  verifica che il file passato come argomento esista (`-e /home/nico/.bashrc`)

- `-b`
  verifica che l'argomento passato sia un block device file (`-b /dev/fd0`). I block device file sono ad esempio i file con cui si accede ai floppy o ai dispositivi cdrom

- `-c`
  verifica che l'argomento passato sia un character device file (`-c /dev/ttyS0`). I character device file sono ad esempio i file con cui si accede al modem o al mouse

- `-d`
  verifica che l'argomento passato sia una directory (`-d /home/nico`)

- `-e`
  verifica che il file passato come argomento esista (`-e /home/nico/.bashrc`)

- `-f`
  verifica che l'argomento passato sia un file regolare (`-f /home/nico/.bashrc`). Sebbene possa sembrare inutile, questo operatore risulta spesso necessario poiché in ambiente *Unix* vale il paradigma *everything is a file* (Ogni cosa è un file). In questo modo, un socket o una periferica, vengono a dialogare con il sistema operativo tramite particolari tipi di file (non regolari)

- `-g`
  verifica che il file passato come argomento abbia il bit relativo al gruppo impostato (`setgid`). Se, ad esempio, una directory ha tale bit impostato, allora ogni file che vi si crea appartiene al gruppo della directory.

- `-h`
  verifica che l'argomento esista e sia un link simbolico

- `-k`
  verifica che il file passato come argomento abbia lo *stiky* bit impostato

- `-p`
  verifica che il file passato sia una pipe

- `-r`
  verifica che il file sia leggibile all'utente che esegue il test (`-r /etc/shadow`)

- `-s`
  verifica che la dimensione del file passato non sia zero (`-s /home/nico/.bashrc`)

- `-t` *n*
  verifica che il file descriptor *n* sia aperto. In genere, i file descriptor aperti sono *stdin*, *stdout* e *stderr* e fanno riferimento ai rispettivi file presenti nella directory `/dev`.

- `-u`
  verifica che il file passato come argomento abbia il bit relativo all'utente impostato (`setuid`). Tale bit viene spesso impostato su eseguibili. Se ad esempio impostassimo il `setuid` su `/sbin/ifconfig`, consentiremmo ad ogni utente della macchina di usarlo con gli stessi privilegi dell'utente *root* (... e sarebbe cosa grave!).

- `-w`
  verifica che il file sia scrivibile dall'utente che esegue il test (`-w /etc/shadow`)

- `-x`
  verifica che il file sia eseguibile dall'utente che esegue il test (`-x /bin/bash`)

- `-G`
  verifica che il file passato come argomento appartenga allo stesso gruppo dell'esecutore del test

- `-L`
  verifica che l'argomento sia un link simbolico (`-L /dev/modem`)

- `-N`
  controlla che il file sia stato modificato dall'ultima lettura

- `-O`
  verifica che il file passato come argomento appartenga all'esecutore del test

- `-S`
  verifica che il file in questione sia un socket (`-S /tmp/mysql.sock`)

- *file1* `-nt` *file2*
  controlla che *file1* sia più nuovo rispetto a *file2*
- *file1* `-ot` *file2*
  controlla che *file1* sia più vecchio rispetto a *file2*
- *file1* `-ef` *file2*
  controlla che *file1* e *file2* rappresentino lo stesso file.

2.4.5 Semplici operatori di Input/Output

Verranno elencati ed introdotti alcuni semplici e fondamentali operatori per gestire l'I/O (Input/Output) di programmi, in modo da rendere più divertenti gli esempi.

| (pipe)

L'operatore | (pipe, tubo) consente di trasformare l'output di un comando in input di un altro. Volendo essere più specifici, si può dire che tale operatore consente di rendere standard input di un programma lo standard output di un altro. L'utilità di tale operatore è inestimabile.

Consideriamo il comando

```
[nico@deepcool nico]$ cat /etc/passwd | grep bash
root:x:0:0:root:/root:/bin/bash
nico:x:500:500:Domenico Delle Side:/home/nico:/bin/bash
```

ed analizziamo cosa è accaduto. `cat` è un programma che stampa sullo standard output (lo schermo nel nostro caso) il contenuto dei file passati come argomento. `grep`, invece, serve per ricercare una particolare espressione all'interno di un testo che può essere contenuto in un file oppure scritto sullo standard input. La sequenza di comandi data, dunque, si può riassumere dicendo che attraverso |, il contenuto del file `/etc/passwd` viene passato a `grep` che cerca poi all'interno del testo la parola `bash`.

<

L'operatore < consente di reindirizzare il contenuto di un file facendolo diventare, ad esempio, standard input di un comando. Il comando

```
[nico@deepcool nico]$ mail nico < /etc/passwd
```

fa comparire per incanto nella mia casella locale di posta il contenuto del file `/etc/passwd`.

> e >>

Gli operatori > e >> svolgono entrambi un compito circa analogo, ambedue infatti si occupano del reindirizzamento di un output. Questi operatori possono essere utilizzati per scrivere il risultato di un comando in un file, tuttavia, presentano una piccola differenza, chiariamo il tutto con un esempio.

```
[nico@deepcool nico]$ man bash > bash
[nico@deepcool nico]$ ls -lh bash
-rw-rw-r--    1 nico     nico         293k Jul 24 19:59 bash
[nico@deepcool nico]$ man bash >> bash
[nico@deepcool nico]$ ls -lh bash
-rw-rw-r--    1 nico     nico         586k Jul 24 19:59 bash
[nico@deepcool nico]$ man bash > bash
[nico@deepcool nico]$ ls -lh bash
-rw-rw-r--    1 nico     nico         293k Jul 24 19:59 bash
```

Analizziamo ciò che è accaduto. Il primo comando `man bash > bash` reindirizza lo standard output del comando `man` all'interno del file `/home/nico/bash`; dato che tale file non esisteva in precedenza, > si occupa di crearlo. Il risultato finale è, come è possibile vedere attraverso il comando `ls -lh`, un file di testo da 293 Kb contenente la pagina manuale di `bash`.
Il comando successivo, `man bash >> bash`, opera lo stesso reindirizzamento, tutavia, constatando che il file `/home/nico/bash` è già esistente, lo apre aggiungendo l'output di `man bash` al testo già presente nel file. Come era ovvio aspettarsi, otteniamo un file di dimensione doppia (586 Kb).
Se eseguiamo nuovamente `man bash > bash`, noteremo che il file originario è stato troncato e poi sovrascritto.

2.4.6 Operatori su bit

Prima di trattare questo argomento, occorre fare una piccola trattazione sulla rappresentazione binaria dell'informazione.

Rappresentazione binaria

All'interno di un calcolatore, l'informazione è rappresentata a livello di tensione che può "accendere" o "spegnere" determinati circuiti. Di conseguenza, ogni informazione può essere convenientemente espressa in termini di due stati che descrivono la situazione in cui si trovano degli ipotetici interruttori che azionano tali circuiti. Per convenzione, si indicano queste configurazioni con **1** (*acceso*) e **0** (*spento*).
Ogni informazione presente in un calcolatore è dunque rappresentata da sequenze di numeri del tipo 1011001011010001, in cui ogni istanza di **1** o **0** rappresenta un *bit* , ovvero l'unità di memoria elementare di un calcolatore. Ad esempio, anche i numeri naturali vengono rappresentati sotto forma binaria (0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011, 4 = 0100, 5 = 0101, 6 = 0110, 7 = 0111, 8 = 1000, 9 = 1001, ecc...).
Gli operatori su bit consentono di eseguire operazioni direttamente su queste quantità. Elenchiamo di seguito quelli offerti dalla shell *Bash*:

- <<
  Operatore di spostamento a sinistra. Viene usato con la sintassi `M << N`, che equivale a dire "sposta di N posti a sinistra i bit che compongono M".
- >>
  Operatore di spostamento a destra. Analogo al precedente, questa volta lo spostamento avviene verso destra.

- &
  Operatore di `AND` su bit. Ritorna un numero binario in cui sono accesi i bit nelle posizioni in cui tutti e due gli argomenti hanno un bit acceso (`1100 & 1001 = 1000`, ovvero `12 & 9 = 8`).

- |
  Operatore di `OR` su bit. Ritorna un numero binario in cui sono accesi i bit nelle posizioni in cui almeno uno degli argomenti ha un bit acceso (`1100 | 1001 = 1101`, ovvero `12 | 9 = 13`).

- ^
  Operatore di `XOR` (`OR` esclusivo) su bit. Ritorna un numero binario in cui sono accesi i bit nelle posizioni in cui gli argomenti hanno bit differenti (`1100 ^ 1001 = 0101`, ovvero `12 ^ 9 = 5`).

Presentiamo ora un simpatico esempio.

Esempio 2.4.3

```
#!/bin/bash
#
# Potenze del due tramite operatori su bit.

EXIT_SUCCESS=0

echo "Inserisci l'esponente della potenza di 2 che vuoi conoscere"
echo "(vale solo per esponenti positivi): "

read ESPONENTE

echo "2^($ESPONENTE) vale: $(( 2 << $(( $ESPONENTE - 1 )) ))"

exit $EXIT_SUCCESS
```

Come funziona:

Lo script sfrutta una nota proprietà dello spostamento di un bit a sinistra, questa operazione, infatti, equivale alla moltiplicazione per due del numero dato. In questo modo è possibile ottenere in maniera molto veloce le potenze di due, spostandosi a sinistra di un numero pari all'esponente diminuito di una unità (il primo due lo abbiamo già, quindi dobbiamo toglierlo).

2.5 Liste di comandi

Uno delle caratteristiche più utili delle shell *Unix* è la possibilità di poter concatenare dei comandi e condizionare agevolmente l'esecuzione di uno in luogo dell'altro. Questa possibilità è offerta da una serie di operatori elencati di seguito.

- ;
  In genere, per separare i comandi impartiti alla shell, è sufficiente scriverne uno per riga; questa convenzione, tuttavia, è piuttosto limitante, poiché potrebbe rendersi necessario impartire più comandi sulla stessa linea. L'operatore `;` serve proprio a questo, consente infatti di eseguire uno dopo l'altro più comandi, ad esempio `cd ; startx`. La shell non si occupa assolutamente dell'esito dei comandi, ma si limita semplicemente ad eseguirli.

- &
  Questo operatore, da usare sempre alla fine di un comando semplice o una lista di comandi, consente di forzarne l'esecuzione in background in una *subshell* (Cfr. sez. 4.7); la shell non aspetta che il comando finisca e gli assegna automaticamente un valore di ritorno pari a 0 (Successo).

- &&
  Questo operatore, in un certo senso, implementa per le liste una sorta di AND logico. Spieghiamoci meglio. Prendiamo in considerazione un caso generico, `comando1 && comando2`, in questo caso comando2 viene lanciato se e soltanto se comando1 è stato eseguito con successo. Il valore di ritorno di questo tipo di lista è quello dell'ultimo eseguito.

- ||
  Anche in questo caso è possibile un parallelismo con gli operatore logici; || rappresenta una sorta di OR per liste. Consideriamo `comando1 || comando2`, in questo caso comando2 verrà eseguito se e soltanto se l'esecuzione di comando1 non andrà a buon fine.Il valore di ritorno di questo tipo di lista è quello dell'ultimo eseguito.

2.6 Uso degli operatori

Dopo aver passato in rassegna i più importanti operatori offerti dalla shell *Bash*, vediamo come utilizzarli con maggiore dettaglio.

Innanzitutto, occorre capire come è possibile valutare una condizione, cioè come eseguire dei test. Niente di più facile, esiste il comando interno test che da alla shell la possibilità di valutare la veridicità della condizione che gli si fornisce. Accanto a test, esiste anche il simbolo [...] che consente alla shell di valutare la condizione interna. Esiste anche una terza via per valutare condizioni ed è fornita da [[...]], analogo ai precedenti e con qualche facilitazione aggiunta (consultare la pagina manuale o il manuale info di *Bash* per maggiori informazioni), tuttavia è presente solo sulle versioni più recenti di *Bash* (dalla versione 2.02). Inoltre, è meglio non mettere troppa carne al fuoco; in questo momento sarete così presi da questa interessantissima ed avvincente guida che rischiereste di bruciarla, un vero peccato!

Come primo esempio, vediamo come è possibilire utilizzare gli operatori logici e le differenze che corrono tra loro. Prima di tutto occorre esaminare come si compongono le condizioni con AND, OR e gli operatori di lista. Un AND logico è vero se e solo se sono vere tutte le condizioni che lo compongono, mentre un OR è falso se e solo se sono false tutte le condizioni che lo compongono (quindi vero negli altri casi).

Esempio 2.6.1

```
#!/bin/bash
#
# Uso degli operatori logici

EXIT_SUCCESS=0

[ -e $HOME/.bashrc ] && echo "Nella mia home è presente .bashrc"
[ ! -e $HOME/non_esisto ] && echo "Il file non_esisto non esiste!"

[ ! -e $HOME/.bashrc ] || echo "Nella mia home è presente .bashrc"
[ -e $HOME/non_esisto ] || echo "Il file non_esisto non esiste!"

exit $EXIT_SUCCESS
```

Come funziona:

Per ogni riga di codice, a meno di situazioni strane, l'esempio stamperà sullo schermo il relativo messaggio. Vediamo in dettaglio cosa accade. `[ -e $HOME/.bashrc ]` verifica la presenza del file `.bashrc` nella home directory di chi lo esegue, poiché questa guida è basata sulla shell *Bash*, è ragionevole supporre che questo file esista, dunque la condizione sarà vera. In questa situazione, l'operatore `&&` si preoccuperà di eseguire il comando seguente `echo ''Nella mia home è presente .bashrc''` , che stamperà sullo schermo il messaggio ritornando un valore di successo. La seconda condizione (`[ ! -e $HOME/non_esisto ]`) valuterà la **non** esistenza, nella home directory dell'esecutore, del file `non_esisto`; non è un nome di file comune quindi non dovrebbe esistere pertanto negando la condizione attraverso l'operatore `!` otterremo una condizione vera e verrà pertanto stampato sullo schermo il messaggio successivo, come prima.

Sappiamo già che in `$HOME` esiste `.bashrc`, dunque il contrario è falso, pertanto nella riga successiva, l'operatore `||` consentirà l'esecuzione del comando successivo, stampando il messaggio sullo schermo. Analogamente, il test successivo è falso e verrà nuovamente stampato un messaggio sullo schermo.

Esempio 2.6.2

```
#!/bin/bash
#
# Uso degli operatori logici (2)

EXIT_SUCCESS=0

[ -e $HOME/.bashrc -a ! -e $HOME/non_esisto ] && \
   echo "Nella mia home esiste .bashrc e non c'è non_esisto"

[ -e $HOME/non_esisto -o -e $HOME/.bashrc ] && \
```

```
    echo "Nella mia home non c'è non_esisto, ma si trova .bashrc"

exit $EXIT_SUCCESS
```

Come funziona:

In questo esempio capiamo la differenza che corre rispettivamente tra gli operatori "-a - &&" e "-o - ||". Infatti, -a e -o sono operatori che collegano logicamente delle condizioni all'interno di un test, mentre && e || collegano "logicamente" dei comandi[10]. In altre parole `-e $HOME/.bashrc -a !  -e $HOME/non_esisto` è una **condizione composta** tramite l'operatore logico -a (AND) e risulta vera quando tutte le *sotto-condizioni* che la compongono sono vere. Allo stesso modo, anche `-e $HOME/non_esisto -o -e $HOME/.bashrc` è una condizione composta, questa volta tramite l'operatore logico -o (OR) e risulta vera quando almeno una delle due *sotto-condizioni* è vera.

Gli operatori && e ||, invece, collegano comandi in base al loro valore di ritorno, come abbiamo visto nell'esempio precedente.

[10] Con la forma [[...]] le cose stanno in maniera leggermente differente. Il lettore interessato può consultare la pagina manuale di *Bash* o il suo manuale info

Capitolo 3

Strutture di controllo

In questo capitolo verranno introdotti i *cicli* e le *istruzioni di selezione*, i mattoni da costruzione di ogni linguaggio di programmazione.

3.1 Cicli

Succede non di rado di dover ripetere un'istruzione nel tempo; questa operazione è fondamentale in ogni progetto di software. La shell *Bash*, mette a disposizione 3 tipi di ciclo differenti: `while`, `for` e `until`.

3.1.1 `while`

Il ciclo `while` consente di eseguire un blocco di istruzioni fino a quando una certa condizione è vera. La verifica della condizione viene effettuata attraverso il comando `test`, oppure racchiudendo la condizione tra parentesi quadre, `[...]`. Un esempio renderà il tutto più chiaro.

Esempio 3.1.1

```
#!/bin/bash
#
# ciclo while e condizioni

EXIT_SUCCESS=0

while [ "$RISPOSTA" != "q" ]
do
      echo "Premi un tasto (per uscire \"q\"):"
      read RISPOSTA
done

# avremmo potuto scrivere equivalentemente:
#
# while test "$RISPOSTA" != "q"
# do
```

```
#       echo "Premi un tasto (per uscire \"q\"):"
#       read RISPOSTA
# done

exit $EXIT_SUCCESS
```

Come funziona:

L'esempio precedente ha introdotto numerose novità. Innanzi tutto abbiamo visto come utilizzare il ciclo `while`:

```
while [ condizione ]
do
      istruzioni
done
```

Usando la lingua parlata, possiamo dire che tale ciclo consiste in "**finché è vera la condizione in parentesi quadre, esegui le istruzioni contenute tra** `do` **e** `done`". Nel caso in esame, la condizione da verificare è `''$RISPOSTA'' != ''q''`. Sottolineamo come prima cosa che le virgolette giocano un ruolo fondamentale, soprattutto quelle relative alla variabile `$RISPOSTA`; se infatti avessimo scritto `$RISPOSTA != ''q''`, la shell avrebbe interrotto l'esecuzione dello script ritornando un messaggio (ed anche un codice[1]) d'errore:

```
[nico@deepcool intro]$ ./script
./script: [: !=: unary operator expected
```

Ciò accade perché all'atto della prima valutazione della condizione presente nel ciclo, la variabile `$RISPOSTA` non è definita, ovvero si ha una condizione del tipo

```
...
while [  != "q" ]
...
```

e la shell non è in grado di interpretarla correttamente. La variabile `$RISPOSTA` ha infatti un valore nullo (*null - value*) e la shell non riesce a valutare la condizione dato che in queste condizioni si aspetterebbe la presenza di un operatore unario (!= non lo è!). Sono fondamentali gli spazi lasciati dopo `[` e prima di `]`, poiché questi consentono alla shell di interpretare correttamente l'uso delle parentesi quadre per valutare una condizione.

Passiamo ora ad analizzare il funzionamento del ciclo. Alla sua prima esecuzione, la condizione `''$RISPOSTA'' != ''q''` è vera poiché la la variabile contiene una stringa vuota, dunque vengono eseguite le istruzioni comprese tra `do` e `done`. Nel nostro caso viene stampato sullo schermo un messaggio e poi si aspetta per acquisire un input dall'utente (`read RISPOSTA`). Acquisito l'input, la condizione viene rivalutata e, qualora risulti nuovamente vera, viene ripetuto il blocco di istruzioni. Non appena l'utente avrà premuto il tasto "q", lo script uscirà dal ciclo ed eseguirà l'istruzione successiva (`exit $EXIT_SUCCESS`).

[1] **Esercizio:** Come si può ricavare il valore di ritorno di un comando?

Per ragioni stilistiche, spesso il ciclo `while` assume una forma lievemente diversa diventando:

```
while [ condizione ]; do
      istruzioni
done
```

Tale scrittura è consentita poiché la shell *Bash* prevede, anche se non lo impone, che ";" funga da separatore di istruzioni. Allo stesso modo, sarebbe perfettamente legale la scrittura

```
while [ condizione ]; do istruzioni; done
```

Si noti inoltre che l'uscita dal ciclo può verificarsi anche quando uno dei comandi interni ad esso riporta un valore di ritorno differente da zero (condizione di errore).

Nella sezione 2.3 abbiamo avuto a che fare con i parametri posizionali e ci siamo posti il problema di come fare a fornirne allo script un numero arbitrario. La questione può essere risolta facendo ricorso al comando interno `shift` .

La sintassi del comando è `shift [N]`, dove `N` è un naturale; con ciò si forza lo spostamente a sinistra della posizione del parametro posizionale di `N` posti. Qualora `N` non fosse passato, si assume che lo spostamento sia di una unità; se invece `N` è zero o un numero maggiore di $#, allora non viene intrapresa alcuna azione sui parametri. L'operazione non modifica il valore di $0. Vediamo quindi come modificare l'esempio 2.3.1.

Esempio 3.1.2

```
#!/bin/bash
#
# Utilizzo dei parametri posizionali (2)

EXIT_SUCCESS=0

TOTALE=$#
echo -n "Hai inserito "

while [ -n "$1" ]; do
    echo -n "$1 "
    shift
done

echo "per un totale di $TOTALE parametri"

exit $EXIT_SUCCESS
```

Come funziona:

Come prima cosa, mostriamo di aver detto la verità:

```
[nico@deepcool nico]$ ./shift.sh param1 param2 param3 param4
Hai inserito param1 param2 param3 param4 per un totale di 4 parametri
```

Lo script non conosceva a priori il numero di parametri inseriti, ma si è comportato in maniera corretta elencando uno ad uno i parametri passati e scrivendone il loro numero.

Analizziamo ora quanto accaduto. Alla prima iterazione del ciclo, viene stampato sullo schermo il valore di `$1` e successivamente eseguita l'istruzione `shift` (Lo spostamento per tanto è di una posizione). In questo modo, a `$#` viene assegnato il valore `$# - 1` ed il vecchio `$2` diventa `$1`. Da ciò si capisce il motivo dell'assegnazione `RITORNO=$#`, se avessimo utilizzato il parametro `$#` dopo l'esecuzione del ciclo per ottenere il numero totale di parametri passati allo script, avremmo ottenuto il valore 0, poiché il contenuto di `$#` sarebbe stato nel frattempo modificato da `shift`.

3.1.2 for

Il ciclo `for` della shell *Bash* è nettamente differente da quello presente in altri linguaggi di programmazione, la sua sintassi è infatti:

```
for ELEMENTO in LISTA
do
    istruzioni
done
```

In lingua corrente potrebbe essere reso con "**per ogni ELEMENTO presente in LISTA esegui i comandi compresi tra** `do` **e** `done`". Possiamo già notare una importante caratteristica; il ciclo `for` consente di definire una variabile, proprio come `read` e l'assegnazione tramite l'operatore =. Consideriamo il seguente esempio.

Esempio 3.1.3

```
#!/bin/bash
#
# Esempio d'uso del ciclo for

EXIT_SUCCESS=0

for file in $(ls $PWD); do
    echo $file
done

exit $EXIT_SUCCESS
```

Come funziona:

All'inizio, il ciclo inizializza una nuova variabile `$file` al primo elemento della lista `$(ls $PWD)`[2] e successivamente stampa sullo schermo il contenuto di tale variabile. Questo processo

[2] La variabile $(ls $PWD) contiene l'output del comando ls $PWD

continua fino a quando non vengono esauriti gli elementi presenti nella lista. La lista di elementi deve essere composta da una serie di parole separate dal separatore (eventualmente più di uno) standard `$IFS`. È possibile definire un separatore standard personalizzato, in modo da processare qualsiasi tipo di lista.

Esempio 3.1.4

```
#!/bin/bash
#
# $IFS personalizzato

EXIT_SUCCESS=0
LISTA_1="uno:due:tre:quattro:cinque:sei:sette:otto:nove:dieci"
LISTA_2="1 2 3 4 5 6 7 8 9 10"
OLD_IFS="$IFS"

export IFS=":"

# Primo ciclo
for i in $LISTA_1; do
    echo $i
done

export IFS="$OLD_IFS"

# Secondo ciclo
for j in $LISTA_2; do
    echo $j
done

exit $EXIT_SUCCESS
```

Come funziona:

L'esempio mostra come sia possibile sfruttare la personalizzazione di `$IFS` per avere maggiore flessibilità nel processare una lista. Dapprima vengono create le liste `$LISTA_1` e `$LISTA_2` mentre la variabile `$OLD_IFS` viene inizializzata al valore di `$IFS`. Successivamente il separatore standard viene cambiato in ":" ed esportato in modo da renderlo disponibile ai processi figli dello script. A questo punto il primo ciclo `for` processa la prima lista separandone gli elementi in base alla presenza di ":" per poi stamparli. All'uscita,`$IFS` viene esportata riassumendo il suo valore originario, in modo da processare la seconda lista e stamparne gli elementi, come nell'esempio precedente.

Ormai dovrebbe essere chiara la differenza tra i parametri `$*` e `$@`. Infatti la prima scrittura fornisce una lista con singolo elemento, mentre la seconda ritorna una lista contenente tutti i parametri passati allo script.

Esempio 3.1.5

```
#!/bin/bash
#
# Uso di $* e $@

EXIT_SUCCESS=0

echo "Inizia il primo ciclo:"
for i in "$*"; do
    echo $i
done
echo -e "Il primo ciclo è finito\n"

echo "Inizia il secondo ciclo:"
for i in "$@"; do
    echo $i
done
echo "Il secondo ciclo è finito"

exit $EXIT_SUCCESS
```

Come funziona:

L'output dello script è più eloquente di ogni spiegazione:

```
[nico@deepcool intro]$ ./script param1 param2 param3 param4 param5
Inizia il primo ciclo:
param1 param2 param3 param4 param5
Il primo ciclo è finito

Inizia il secondo ciclo:
param1
param2
param3
param4
param5
Il secondo ciclo è finito
```

Con oppurtuni accorgimenti, possiamo ottenere un ciclo `for` analogo a quello del *C*.

Esempio 3.1.6

```
#!/bin/bash
#
# un for da C!
```

```
EXIT_SUCCESS=0

for i in $(seq 1 10); do
    echo $i
done

exit $EXIT_SUCCESS
```

Come funziona:

Sfruttando l'utility seq (man seq per fugare ogni dubbio relativo al suo utilizzo) siamo riusciti ad ottenere un ciclo `for` che itera le sue istruzioni alla stessa maniera del `for` del *C*; `seq 1 10` infatti restituisce una lista contenente i numeri da 1 a 10 (estremi inclusi).

3.1.3 `until`

Il ciclo `until` è simile al `while`,

```
until [ CONDIZIONE ]
do
      COMANDI
done
```

l'unica differenza risiede nel fatto che i comandi compresi tra do e done vengono ripetuti fino a quando la condizione che si verifica è falsa. In lingua italiana, il ciclo `until` suonerebbe come: **fino a quando CONDIZIONE non è vera esegui COMANDI**. Consideriamo l'Esempio 3.1.1 modificandolo opportunamente.

Esempio 3.1.7

```
#!/bin/bash
#
# ciclo until e condizioni

EXIT_SUCCESS=0

until [ "$RISPOSTA" = "q" ]; do
      echo "Premi un tasto (per uscire \"q\"):"
      read RISPOSTA
done

exit $EXIT_SUCCESS
```

Come funziona:

Il comportamento dello script è del tutto analogo a quello dell'Esempio 2.4.1, in questo caso, però, l'iterazione avviene utilizzando un ciclo `until`, per questo motivo la condizione da valutare è stata cambiata. Mentre in 2.4.1 questa era `‘‘$RISPOSTA’’ != ‘‘q’’`, nel presente deve essere negata in modo da funzionare correttamente. Si ottiene dunque `‘‘$RISPOSTA’’ = ‘‘q’’`.

3.2 Istruzioni di selezione

Le istruzioni di selezione, come i cicli, servono a controllare l’esecuzione di blocchi di codice.

Abbiamo visto in precedenza che il contenuto di un ciclo viene eseguito a seconda della veridicità della condizione; un comportamento del tutto analogo si applica anche in questo caso, con la sola differenza che non esiste alcuna iterazione del codice interno all’istruzione.

La selezione è utile, ad esempio, tutte quelle volte in cui si deve prendere una decisione a seconda del valore di una variabile o altra condizione.

3.2.1 if

L’istruzione `if` consente di eseguire un blocco di codice se una condizione è vera. Come è da aspettarsi, la condizione può essere valutata sia con `[ ... ]`, sia con `test` [3]. Come da nostra tradizione, consideriamo un semplice esempio per illustrarne il funzionamento:

Esempio 3.2.1

```
#!/bin/bash
#
# Un semplice esempio d'uso per if

EXIT_SUCCESS=0
EXIT_FAILURE=1

echo "Ti sta piacendo la guida? (si/no)"

read RISPOSTA

if [ "$RISPOSTA" != "si" -a "$RISPOSTA" != "no" ] ; then
   echo "Rispondi solo con un si o con un no"
   exit $EXIT_FAILURE
fi

if [ "$RISPOSTA" == "no" ] ; then
   echo "Vai in castigo!"
   exit $EXIT_FAILURE
fi

echo "Tu si che sei un bravo ragazzo!"
```

[3] A dire il vero, esistono anche altri modi per dare condizioni in pasto ad `if`, ma chiariremo tutto in seguito.

```
exit $EXIT_SUCCESS
```

Come funziona:

Procediamo con l'analisi dell'esempio riportato. Come prima cosa, vengono definite due fariabili `$EXIT_SUCCESS` ed `$EXIT_FAILURE` che serviranno a specificare il valore di ritorno del nostro script alla shell. Successivamente viene presentata una domanda alla quale occorre rispondere con un si o cono un no. Il primo controlla che `$RISPOSTA` non sia diversa da "si" o "no". La condizione tra `[ ... ]` è infatti un `AND` logico e risulta verà se e solo se sono vere le due condizioni che la compongono. La condizione è dunque vera solo quando risposta è diversa sia da "si", sia da "no". In tale evenienza, viene eseguito il codice interno, ovvero viene stampato un messaggio di cortesia (`Rispondi solo con un si o con un no`) e si termina l'esecuzione ritornando un codice d'errore.

Qualora si fosse risposto correttamente, la shell determina quale sia stata la risposta; il secondo `if`, infatti, serve a verificare se la risposta è stata un no, in tal caso stampa sullo schermo un cortese invito (!) all'esecutore e ritorna all'ambiente d'esecuzione un codice d'errore. Se, invece, la risposta non è "no", allora sarà sicuramente un "si" (non c'è altra possibilità), perciò verrà scritto sullo schermo che chi l'ha eseguito è un bravo ragazzo (!) e lo script terminerà ritornando un valore di successo.

Anche qui vediamo di sintetizzare e rendere in lingua italiana l'istruzione.

```
if [ CONDIZIONE ] ; then
   COMANDI
fi
```

Pertanto, possiamo "leggere" un `if` come **Se è vera CONDIZONE, allora esegui tutti i COMANDI compresi tra** `then` **e** `fi`.

3.2.2 `if - else`

Può accadere di arrivare ad un punto in uno script in cui occorre fare due operazioni differenti a seconda che una certa condizione sia vera o meno. Ciò è simile a quanto fa ogni persona quando si trova in un punto in cui la strada si biforca e deve decidere dove andare. Ad esempio, un automobilista potrebbe fare questo ragionamento "*Se andando a sinistra giungo a destinazione, allora procedo in quella direzione, altrimenti giro a destra*".

Riconsideriamo l'esempio 3.2.1 e proviamo a scriverlo in modo diverso.

Esempio 3.2.2

```
#!/bin/bash
#
# Un semplice esempio d'uso per if - else
```

```
EXIT_SUCCESS=0
EXIT_FAILURE=1

echo "Ti sta piacendo la guida? (si/no)"

read RISPOSTA

if [ "$RISPOSTA" != "si" -a "$RISPOSTA" != "no" ] ; then
   echo "Rispondi solo con un si o con un no"
   exit $EXIT_FAILURE
fi

if [ "$RISPOSTA" == "no" ] ; then
   echo "Vai in castigo!"
   exit $EXIT_FAILURE
else
   echo "Tu si che sei un bravo ragazzo!"
   exit $EXIT_SUCCES
fi
```

Come funziona:

Come possiamo notare, l'unica cosa ad esser cambiata è il secondo `if`, che ora contiene l'istruzione `else`. In questo modo, qualora `$RISPOSTA` non fosse "no", sarebbe alternativamente eseguito il codice compreso tra `else` e `fi`. Riassumendo, un costrutto `if - else`

```
if [ CONDIZIONE ] ; then
   COMANDI1
else
   COMANDI2
fi
```

nella lingua di Dante suonerebbe così: **Se è vera CONDIZONE, allora esegui COMANDI1 (presenti tra `then` ed `else`), altrimenti esegui COMANDI2 (presenti tra `else` e `fi`).**

3.2.3 `if - elif - else`

Nella sezione precedente (3.2.2) abbiamo lasciato un automobilista per strada, mentre decideva se girare a sinistra o a destra. Non vogliamo che rimanga fermo lì per sempre, perciò cerchiamo di farlo andare avanti.
Supponiamo che alla fine abbia deciso di procedere a destra e che ora si trovi ad un incrocio; in questo caso non dovrà solo scegliere la strada che lo porterà a destinazione, ma anche quella più breve.
Ragionare in termini di `if - else` diventà un po' più complicato e si ha bisogno di un nuovo costrutto che semplifichi la vita (forse all'automobilista basterebbe una cartina stradale). Siamo fortunati, la shell *Bash* è prodiga nei nostri confronti e ci fornisce il costrutto `if - elsif - else`. Vediamo come utilizzarlo rivisitando l'esempio precedente.

Esempio 3.2.3

```
#!/bin/bash
#
# Un semplice esempio d'uso per if - elif - else

EXIT_SUCCESS=0
EXIT_FAILURE=1

echo "Ti sta piacendo la guida? (si/no)"

read RISPOSTA

if [ "$RISPOSTA" == "si" ] ; then
   echo "Tu si che sei un bravo ragazzo!"
   exit $EXIT_SUCCESS
elif [ "$RISPOSTA" == "no" ] ; then
   echo "Vai in castigo!"
   exit $EXIT_FAILURE
else
   echo "Rispondi solo con un si o con un no"
   exit $EXIT_FAILURE
fi
```

Come funziona:

L'esempio è chiaro, si analizza la prima condizione (`‘‘$RISPOSTA’’ == ‘‘si’’`), se risultasse vera, lo script eseguirebbe i comandi compresi tra `then` ed `elif`, altrimenti passerebbe a valutare la condizione seguente (`‘‘$RISPOSTA’’ == ‘‘no’’`). Nuovamente, se questa fosse vera, verrebbero eseguite le istruzioni presenti tra `then` ed `else`, in caso contrario (entrambe le condizioni precedenti false) sarebbero eseguiti i comandi tra `else` e `fi`. Facile, vero?

3.2.4 case

Insistiamo con il nostro esempio; lo stiamo presentando in diverse salse, non diventeremo dei cuochi per questo, ma di sicuro impareremo ad usare meglio la shell. Questa volta utilizzeremo il costrutto case, molto utile quando si vuole confrontare una variabile con un possibile insieme di valori ed agire in modo differente a seconda del valore che questa ha assunto.

Esempio 3.2.4

```
#!/bin/bash
#
# Uso di case

EXIT_SUCCESS=0
EXIT_FAILURE=1
```

```
echo "Ti sta piacendo la guida?"

read RISPOSTA

case $RISPOSTA in
  si|s|yes|y)
    echo "Tu si che sei un bravo ragazzo!"
    exit $EXIT_SUCCESS
    ;;
  no|n)
    echo "Vai in castigo!"
    exit $EXIT_FAILURE
    ;;
  *)
    echo "Non ho capito, puoi ripetere?"
    exit $EXIT_FAILURE
    ;;
esac
```

Come funziona:

Dopo aver letto l'input dell'utente, `$RISPOSTA` viene passata al costrutto `esac`, che la analizza. Ci accorgiamo subito che questa volta lo script è più elastico, infatti non chiede di scrivere un "si" o un "no", ma lascia ampia libertà all'utente. Se `$RISPOSTA` è uno tra "`si s yes y`", allora lo script eseguirà il blocco interno fino alla prima occorrenza di "`;;`". Come si può notare, per separare una lista di possibili valori assunti da `$RISPOSTA` viene usato `|`, da non confondersi con l'operatore. Qualora i valori elencati nella prima possibilità non corrispondessero con la variabile in esame, si passerebbe ad analizzare la seconda ("`no n`"). Nuovamente, in caso di esito positvo, verrebbe eseguito il codice interno, sino alla prima occorrenza di "`;;`". Se nessuna delle possibilità corrispondesse con `$RISPOSTA`, allora verrebbe eseguito il codice compreso tra "`*)`" e l'ultima occorrenza di "`;;`", che rappresenta in un certo senso il l'azione di default, dato che `*` un *pattern*[4] che individua qualunque valore assunto da `$RISPOSTA`. Il costrutto si conclude con l'istruzione `esac`.

Dopo aver visto `case` in azione, cerchiamo di generalizzare ciò che abbiamo imparato. `case` controlla sequenzialmente una variabile in un elenco di casi forniti dall'utente. Un caso può essere composto da più possibilità, separate dal simbolo `|`. Ogni caso va concluso con una parentesi tonda di chiusura "`)`". Il primo caso che soddisfa la variabile data provoca l'uscita dalla struttura, pertanto si deve evitare di porre come primo caso un valore molto generico, ma elencare in primo luogo i casi particolari. Occorre soffermarsi ulteriormente sui casi; questi infatti possono anche essere presentati sotto forma di semplici espressioni regolari. Ad esempio, se volessimo controllare una variabile per scoprire se questa contiene una lettera maiuscola o una minuscola oppure un numero, potremmo scrivere:

[4] Si consulti a riguardo [3].

```
...

  [a-z]) echo "Hai premuto una lettera minuscola" ;;
  [A-Z]) echo "Hai premuto una lettera maiuscola" ;;
  [0-9]) echo "Hai premuto un numero" ;;

...
```

L'espressione `[a-z]` (`[A-Z]`) sta per "tutte le lettere comprese tra a e z (A e Z)", così pure `[0-9]` sta per "tutti i numeri compresi tra 0 e 9". Il codice precedente ci ha anche mostrato che è legale allineare caso, comandi (eventualmente separati da "`;`") e terminatore ("`;;`"). Il default rappresenta un caso particolare di espansione, la shell *Bash*, infatti, interpreta "*" come un qualsiasi valore.

Ogni blocco di codice da eseguire, deve essere seguito da "`;;`", che ne indica la fine, mentre la conclusione dell'istruzione è segnata dalla parola chiave `esac`.

3.3 Un ibrido: `select`

FIXME: Inserire paragrafo appena possibile

Capitolo 4

Argomenti avanzati

In questo capitolo sono raggruppati alcuni argomenti più specialistici che semplificano la vita di chi si accinge a fare uno script con la shell *Bash* (Alzi la mano chi non è un po' pigro!).

4.1 Funzioni

Immaginiamo di aver fatto uno script in cui le stesse istruzioni dovrebbero essere scritte più volte all'interno del codice. Il *copia ed incolla* potrebbe essere una soluzione a questo fastidiosissimo problema, ma di sicuro non sarebbe elegante.

Cerchiamo di risolvere il problema pensando a ciò che facciamo durante il quotidiano. Quando andiamo a fare la spesa e compriamo divesi oggetti, mettiamo tutto all'interno di un sacchetto, anziché portare ogni cosa a mano; ecco il lampo di genio, ci serve qualcosa di analogo!

La shell *Bash* ci offre degli speciali sacchetti per la spesa, le **funzioni** (Saranno biodegradabili?) con le quali possiamo risparmiare molta fatica e conservare uno stile di programmazione elegante. La definizione schematica di una funzione *Bash* è

```
[ function ] nome_funzione () {
    COMANDI
}
```

Analizziamo brevemente quanto scritto; la parola chiave `function` è tra parentesi quadre poiché opzionale, infatti possiamo equivalentemente scrivere

```
function nome_funzione () {
    COMANDI
}
```

o

```
nome_funzione () {
    COMANDI
}
```

L'unica differenza tra le due forme dichiarative sta nel fatto che la scrittura di `function` consente di omettere le parentesi tonde `()`, altrimenti obbligatorie. Altra cosa importante da notare è che le parentesi graffe devono essere separate dal testo che le precede tramite uno spazio, un `TAB` (`\t`) o un `NEWLINE` (`\n`).

Definendo una funzione, si ha l'opportunità di richiamare i comandi in essa contenuti in maniera più semplice all'interno di uno script, possiamo anche passarle argomenti sotto forma di parametri posizionali (sezione 2.3) e ritornare allo script un valore attraverso il quale scegliere cosa fare successivamente[1]. Nel seguito analizzeremo altre peculiarità delle funzioni.
È ora di smettere con le parole e considerare un esempio.

Esempio 4.1.1

```
#!/bin/bash
#
# Un primo esempio di funzione

EXIT_SUCCESS=0
EXIT_FAILURE=1
CONTINUA=2

fai_questo () {

  echo "Hai scritto: $1"
  echo "Vuoi smettere?"
  read RISPOSTA

  case "$RISPOSTA" in
    si|s|yes|y)
      return $EXIT_SUCCESS
      ;;
    no|n)
      return $CONTINUA
      ;;
    *)
      echo "Non ho capito cosa hai scritto, termino subito!"
      exit $EXIT_FAILURE
      ;;
  esac

}


while true; do
```

[1] Qualora non venga modificato esplicitamente tramite l'uso del comando interno `return`, il valore di ritorno di una funzione è quello dell'ultimo comando eseguito al suo interno.

```
  echo "Scrivi una parola: "
  read PRESS

  fai_questo $PRESS
  RITORNO=$?

  case $RITORNO in
    $EXIT_SUCCESS)
      exit $RITORNO
      ;;
    $CONTINUA)
      :
      ;;
  esac
done
```

Come funziona:

Abbiamo a che fare con la nostra prima funzione, ovvero con `fai_questo()`, che utilizziamo per raggruppare una serie di comandi.

Procediamo con ordine, innanzi tutto vediamo che la funzione fa uso del parametro posizionale `$1`, ciò vuol dire che `fai_questo()` accetta un solo argomento che si passa alla funzione sotto forma di parametro posizionale. Successivamente viene letto un valore dalla tastiera ed in base alla risposta data si ritorna, attraverso l'istruzione interna `return`, un determinato valore o si conclude lo script nel caso in cui la risposta non fosse chiara.

Finito il corpo della funzione, abbiamo il resto dello script che consiste in un ciclo infinito, `while true; do ...`. Il comando `true` (Del pacchetto *GNU Shell Utils*) non compie assolutamente alcuna azione, ma ritorna all'ambiente di esecuzione un codice di successo, quindi il ciclo continua indefinitamente. L'uscita o meno dal ciclo dipende dal valore di ritorno della funzione `fai_questo()`. Se il ritorno è `$EXIT_SUCCESS`, allora viene interrotto non solo il ciclo, ma anche l'esecuzione dello script tramite la chiamata `exit $RITORNO`. Se invece il ritorno è `$CONTINUA`, allora viene eseguito il *comando* `:`, che non è altro se non l'equivalente di `true` interno alla shell.

4.1.1 Funzioni e variabili locali

Può accadere che sia necessario utilizzare all'interno di una funzione un nome di variabile già utilizzato nel codice dello script. Ciò accade spesso negli script di grosse dimensioni, in cui si vuole che le variabili abbiano nomi espressivi.

Per evitare di modificare il valore della variabile globale (Globale? Si, rispetto a quella presente nella funzione, ma in realtà si tratta sempre di una variabile la cui esistenza è limitata all'esecuzione dello script), la shell *Bash* consente di dichiarare delle variabili che "vivono" solo all'interno del corpo della funzione, tali variabili vengono dette **locali** e vengono dichiarate attraverso l'istruzione `local [VARIABILE[=VALORE]]`.

Come al solito, le parentesi quadre rappresentano argomenti facoltativi, infatti `local` può essere richiamato da solo (sempre all'interno di una funzione) con il risultato di stampare sullo standard output una lista di variabili locali definite in una funzione. Vediamo di chiarire le nostre idee con un esempio.

Esempio 4.1.2

```
#!/bin/bash
#
# Uso di local

EXIT_SUCCESS=0

funzione () {

    local VARIABILE="Sono all'interno della funzione"

    echo -e "\t$VARIABILE"

}

VARIABILE="Sono all'esterno della funzione"

echo $VARIABILE

funzione

echo $VARIABILE

exit $EXIT_SUCCESS
```

Come funziona:

Dopo i soliti convenevoli, abbiamo definito la funzione `funzione()` (Bel nome, vero?) ed all'interno di questa la variabile locale `$VARIABILE` (Quelli che usiamo, sono sempre nomi espressivi!), assegnadole il valore "`Sono all'interno della funzione`". Nel seguito, definiamo per tutto lo script la variabile `$VARIABILE` (locale allo script, ma globale rispetto alla funzione) assegnandole il valore "`Sono all'esterno della funzione`". Se eseguiamo lo script, ci accorgiamo che l'assegnazione fatta all'interno di `funzione()` non intacca minimamente il valore "globale" di `$VARIABILE`, infatti otteniamo:

```
[nico@deepcool nico]$ ./local.sh
Sono all'esterno della funzione
        Sono all'interno della funzione
Sono all'esterno della funzione
```

4.1.2 Funzioni ricorsive

La shell *Bash* consente di scrivere funzioni che richiamano sè stesse. Tali funzioni vengono dette **ricorsive**. Una funzione ricorsiva si rende particolarmente utile quando all'interno di una funzione occorre applicare ad un dato la funzione stessa, in questo modo, anziché realizzare complessi passaggi di valori, si può semplicemente risolvere il problema consentendo alla funzione di richiamarsi.

Chiariamo il tutto con un esempio. Capita spesso, scambiando file con amici, di averne alcuni con estensioni in maiuscolo (`file.JPG`, `file.HTM`) sparsi per il proprio disco fisso. Sarebbe bello escogitare un sistema per far cambiare automagicamente tutte queste estensioni in minuscolo. Proviamo a vedere come fare!

Esempio 4.1.3

```
#!/bin/bash
#
# Una funzione ricorsiva

EXIT_SUCCESS=0
EXIT_FAILURE=1

utilizzo () {

    echo "$(basename $0) <estensione> [directory di partenza]"
    exit $EXIT_FAILURE

}

sostituisci_estensione () {

    E=$1
# Un'abbreviazione per ESTENSIONE

    if [ -z "$2" ]; then
        DIR=$PWD
    else
        DIR=$2
    fi

    N_E=$(echo $E | tr [A-Z] [a-z])
# Un'abbreviazione per NUOVA_ESTENSIONE

    cd $DIR

    for ELEMENTO in $(ls) ; do
        if [ -d "$ELEMENTO" ]; then
```

```
            sostituisci_estensione "$E" "$ELEMENTO"
        elif [ -n "$(echo $ELEMENTO | grep -E "\.${E}$")" ]; then
            mv $ELEMENTO ${ELEMENTO%$E}$N_E
        fi
    done

    cd ..

}

if [ $# -eq 0 -o $# -gt 2 ]; then
    utilizzo
# Se lo script è stato lanciato con 0 o più di 2 argomenti,
# mostriamo come si lancia e ritorniamo un errore.
fi

echo "INIZIO QUI!"

sostituisci_estensione "$1" "$2"

echo "HO FINITO!"

exit $EXIT_SUCCESS
```

Come funziona:

Come prima cosa, definiamo una funzione `utilizzo()` che ci sarà utile per dire con cortesia che il programma è stato utilizzato male. Per ottenere il nome dello script, utilizziamo il comando basename sul parametro posizionale `$0`. Successivamente, definiamo la funzione `sostituisci_estensione()`; questa accetta 2 valori, il primo è l'estensione (in maiuscolo!) che vogliamo sostituire, il secondo è, invece, la directory dalla quale partire ad applicare lo script (qualora non si fornisse il secondo valore, la directory di partenza sarebbe `$PWD`). Questi valori vengono poi assegnati rispettivamente alle variabili `$ESTENSIONE` e `$DIR`. Tramite il comando `tr` (guardarne la pagina manuale!!) si ottiene l'estensione in caratteri minuscoli e la si deposita nella variabile `$NUOVA_ESTENSIONE`.

Ora inizia il lavoro sporco! Si entra nella directory di partenza (`cd $DIR`) e si analizzano ad uno ad uno i file presenti. Se uno di questi è una directory, allora gli si riapplica la funzione (`sostituisci_estensione $ESTENSIONE ''$ELEMENTO''`), altrimenti si verifica che il file abbia l'estensione data (`echo $ELEMENTO | grep -E ''\.${ESTENSIONE}$''`), `man grep` e `man 7 regex` per informazioni su grep e le espressioni regolari). Se il file ha l'estensione cercata, allora viene fatto il cambio, altrimenti si prosegue con il ciclo. Una volta finiti gli elementi da controllare, si ritorna nella directory superiore. Qui finisce la funzione ricorsiva.

Il resto dello script non fa altro che controllare che siano stati forniti da uno a due parametri, per poi, in caso affermativo, richiamare la funzione `sostituisci_estensione()` sui parametri passati.

Come ulteriore esempio sulle funzioni ricorsive, consideriamo una implementazione dell'*algoritmo di Euclide* per il calcolo del **massimo comun divisore** (`mcd`) di due numeri. Questo algoritmo consiste nella relazione per ricorrenza (valida per m ed n naturali):

$$\begin{cases} \mathtt{mcd}(0,n) & = n \\ \mathtt{mcd}(m,n) & = \mathtt{mcd}(n \bmod m, m) \qquad \text{per} \quad m > 0 \end{cases}$$

dove con `mod` abbiamo indicato genericamente l'operatore %.

Esempio 4.1.4

```
#!/bin/bash
#
# Funzioni ricorsive: calcolo del massimo comun divisore
# di due numeri tramite l'algoritmo di Euclide

EXIT_SUCCESS=0
EXIT_FAILURE=1

utilizzo () {

    echo "$(basename $0) <primo numero> <secondo numero>"
    exit $EXIT_FAILURE

}

mcd () {

    m=$1
    n=$2

    if [ $n -eq 0 -a $m -eq 0 ]; then
        echo "mcd(0,0) non è definito!"
        exit $EXIT_FAILURE
    elif [ $m -eq 0 ]; then
        return $n
    elif [ $n -eq 0 ]; then
        return $m
    fi

    mcd $(( $n % $m )) $m

}
```

```
if [ $# -ne 2 ]; then
    utilizzo
fi

mcd $1 $2

MCD=$?

echo "Il massimo comun divisore dei numeri \"$1\" \"$2\" è: $MCD"

exit $EXIT_SUCCESS
```

Come funziona:

L'implementazione consiste in un'applicazione ricorsiva della funzione `mcd()` fino a quando `$n` o `$m` risulti essere uguale a zero, in tale caso, infatti, la relazione di ricorrenza ci dice che il calcolo del mcd è semplicissimo. La funzione controlla, dunque, che `$n` o `$m` non valgano entrambe zero (`mcd(0,0)` non è definito) e continua effettuando dei test su `$n` e `$m`. Se una di queste vale zero, si ritorna il valore dell'altra.

Qualora non si fosse in uno di questi casi, la funzione si richiama, applicandosi ai valori "`$n % $m`" e `$m` fino a quando non si giunge al caso semplice (`$n` o `$m` uguali a zero). In ultimo, trovato il mcd dei due numeri, la funzione ne fa il suo valore di ritorno, infatti, per ottenerlo abbiamo usato l'assegnazione `MCD=$?`.

4.2 Array o vettori

Nella sezione 2.2 abbiamo visto che le variabili sono dei contenitori in cui conserviamo valori. Continuando con la nostra analogia, se andassimo in un negozio di casalinghi, potremmo acquistare dei set di contenitori, che, magari, potrebbero essere legati l'uno all'altro: questi sono gli **array** o **vettori** (Noi cercheremo di chiamarli sempre vettori).

Per il momento cerchiamo di capire meglio cosa siano i vettori e per farlo, aiutiamoci con la figura 4.1. Nella figura schematizziamo una variabile come un "contenitore singolo" (`VARIABILE`), mentre un vettore viene rappresentato come una sequenza ordinata di contenitori caratterizzata dall'avere "cognome" e "nome". Spieghiamo subito cosa si intende con cognome e nome: tutti gli elementi del vettore hanno in comune la parola `ARRAY` (il cognome) e vengono distinti l'uno dall'altro attraverso un indice sequenziale (il nome) che parte sempre da zero. Se qualcuno, guardando la figura, vi dicesse ≪Ti presento "`ARRAY[0]`"!≫ (Nome: 0, Cognome: `ARRAY`), non sarebbe in vena di scherzi, ma vi starebbe solo indicando il primo elemento del vettore `ARRAY`.

Ritorniamo a parlare della shell e chiariamo subito che *Bash*, al momento della scrittura, mette a disposizione soltanto vettori monodimensionali, quindi non dobbiamo aspettarci di

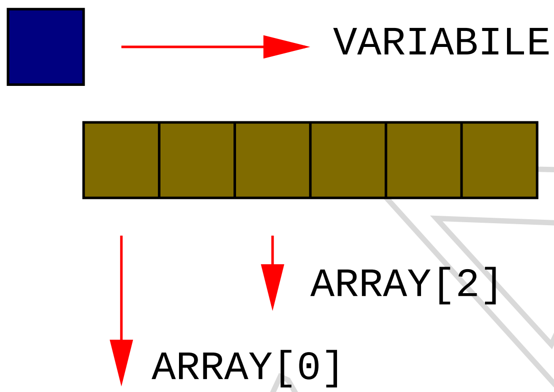

Figura 4.1: Array e variabili

avere a che fare con oggetti del tipo `ARRAY[0][1]`. Forse in futuro, oltre a nome e cognome, ci sarà anche il codice fiscale ed avremo dunque vettori bidimensionali.

Consideriamo qualcosa di più specifico. *Bash* consente di dichiarare esplicitamente un vettore attraverso il comando `declare -a`, tuttavia qualunque riga di codice del tipo `ARRAY[INDICE]=VALORE` creerà automaticamente un vettore. Un'altra forma legale per dichiarare un vettore è l'assegnazione esplicita di tutti i suoi elementi nella forma: `ARRAY=(VALORE_1 VALORE_2 ... VALORE_N)`.
OCCORRE FARE MOLTA ATTENZIONE AL FATTO CHE L'INDICE DI UN VETTORE PARTE SEMPRE DA ZERO!

Riassumendo, creiamo un vettore in questi modi:

```
# Usando declare
declare -a VETTORE
declare -a ALTRO_VETTORE[10]

# Nel secondo caso abbiamo creato un vettore di 10 elementi, il
# primo è ALTRO_VETTORE[0], mentre l'ultimo ALTRO_VETTORE[9]


# Mediante assegnazione di un elemento, in questo caso il primo
# per ricordare che L'INDICE PARTE DA 0!
VETTORE[0]=10


# Attraverso l'assegnazione di tutti i suoi elementi
VETTORE=(11 24 32 88 55)

# Provate a giocare la cinquina!
```

Ora che abbiamo capito come dichiarare un vettore, dobbiamo anche vedere come poter ottenere i valori in esso contenuti. Per evitare di dar luogo ad ambiguità, occorre sempre utilizzare la forma `${VETTORE[INDICE]}`, in questo modo la shell capirà che stiamo facendo riferimento all'*(INDICE+1)-esimo* elemento del vettore `VETTORE` e lo sostituirà con il suo valore, proprio come accade con `${VARIABILE}`.

Esempio 4.2.1

```
#!/bin/bash
#
# uso dei vettori

declare -a VETTORE[3]

VETTORE[0]="Tutto è andato bene"
VETTORE[1]="Hai inserito un numero maggiore di 20"
VETTORE[2]="Hai inserito un numero minore di 10"

inserisci_numero () {

    echo "Inserisci un numero compreso tra 10 e 20"

    read NUMERO

    if [ "$NUMERO" -lt 10 ]; then
        return 2
    elif [ "$NUMERO" -gt 20 ]; then
        return 1
    else
        return 0
    fi

}

inserisci_numero
RITORNO=$?
echo ${VETTORE[$RITORNO]}

exit $RITORNO
```

Come funziona:

All'inizio del codice, dichiariamo tramite `declare` il vettore a 3 elementi `VETTORE` e successivamente lo popoliamo con delle stringhe. Proseguendo definiamo la funzione `inserisci_numero()` alla quale diamo il compito di leggere dalla tastiera un numero e di stabilire se questo è compreso tra 10 e 20 o meno. Qualora il numero inserito non soddisfacesse tale richiesta, la funzione ne stabilirebbe il perché e ritornerebbe un opportuno valore per caratterizzare l'evento.

In ogni caso, il valore di ritorno della funzione viene immagazzinato nella variabile `$RITORNO` che viene usata come indice per il vettore `VETTORE`. In base al valore di `$RITORNO`, tramite la linea di codice `echo ${VETTORE[$RITORNO]}` viene stampato sullo schermo un messaggio a seconda dell'esito delle operazioni effettuate da `inserisci_numero()`.

Continuiamo ad esaminare l'argomento "vettori". È possibile utilizzare come indice di un vettore i caratteri "*" e "@"; la scrittura `${VETTORE[*]}` espande il vettore in un'unica stringa composta dagli elementi del vettore separati dal primo carattere presente in `$IFS`, mentre `${VETTORE[@]}` espande ogni elemento in una stringa separata. Alcuni delle tecniche di espansione di parametri possono essere applicate con successo non solo ad ogni elemento di un vettore, ma anche all'intero vettore stesso. Ad esempio, `${#VETTORE[*]}` e `${#VETTORE[@]}` possono essere usati per ottenere il numero di elementi presenti in un vettore. **FIXME: Inserire una spiegazione dettagliata a riguardo**

4.3 Reindirizzamento dell'Input/Output

Nella sezione 2.4.5 abbiamo già avuto a che fare con gli operatori di input/output (Nel seguito abbreviato con "I/O"); in particolare, ci siamo soffermati su "|", "<" e la coppia "> - >>". In questa sezione, cercheremo di estendere ciò che già conosciamo, introducendo alcuni nuovi concetti.

Prima che un comando venga eseguito, è possibile redirigere sia il suo canale di input che quello di output, utilizzando gli operatori di reindirizzamento. Il reindirizzamento può essere anche utilizzato per chiudere ed aprire file. Il reindirizzamento ha una sintassi bene precisa, che la shell interpreta da sinistra verso destra; inoltre, dato che questo può avvenire utilizzando più di una espressione, è importante l'ordine con cui queste vengono impartite alla shell.

4.3.1 I file in un sistema Unix

Ci si può chiedere perché dedicare addirittura un'intera sezione a questo argomento, che all'apparenza sembrerebbe piuttosto semplice. In un sistema *Unix*, tuttavia, i file occupano un ruolo assai importante; questi, infatti, forniscono un'interfaccia semplice e consistente ai vari servizi del sistema operativo ed ai dispositivi in genere. Finalmente, capiamo il significato del principio "*everything is a file*" a cui abbiamo fatto riferimento nella sezione 2.4.4.

Ciò significa che i programmi, in generale, possono usare file su disco, porte seriali, connessioni di rete, stampanti ed altri dispositivi tutti nello stesso modo. Nella realtà, dunque, quasi tutto è reso disponibile sotto forma di file o in alcuni casi file particolari, con differenze minime rispetto al caso generale, in modo da non violare il principio espresso in precedenza.

Prima di procedere, occorre introdurre una nozione fondamentale. Un **descrittore di file** è una entità (Nel nostro caso, la shell, un numero) che identifica in maniera univoca un file aperto in una particolare modalità, quale ad esempio lettura o scrittura. In questo modo, ogni azione su un file può essere intrapresa agendo sul suo descrittore.

Se non istruita diversamente, ad esempio, per ogni processo la shell associa sempre dei descrittori di file allo standard input (**Descrittore 0**), standard output (**Descrittore 1**) e standard error (**Descrittore 2**).

4.3.2 Apertura di descrittori di file in lettura/scrittura

La shell *Bash* consente l'apertura di descrittori di file personalizzati, in modo da gestire in maniera efficiente l'I/O nei propri script, utilizzando ulteriori descrittori oltre a 0, 1 e 2. Per effettuare questa operazione si utilizza l'operatore `<>`, la sintassi è la seguente:

```
[n]<>parola
```

L'operatore `<>` causa l'apertura in lettura/scrittura del file identificato dall'espansione di `parola` sul descrittore di file `n`. Se non è passato alcun descrittore di file, la shell assume che il descrittore di file sia `0`. Qualora il file non esistesse, verrebbe creato.

4.3.3 Reindirizzamento dell'Input

La sintassi generale per effettuare un reindirizzamento dell'input è:

```
[n]<parola
```

Anche qui, il parametro opzionale `n` rappresenta il descrittore di file su cui indirizzare il risultato dell'espansione di `parola`. Nel caso in cui non venisse fornito il descrittore di file `n`, il reindirizzamento viene effettuato su descrittore 0, ovvero lo standard input.

Ad esempio, quando abbiamo visto il comando `mail nico < /etc/passwd`, non avendo specificato alcun descrittore di file, intendendevamo redirigere il contenuto di `/etc/passwd` sullo standard input del comando `mail nico`.

4.3.4 Reindirizzamento dell'Output

Analogamente ai casi precedenti, la sintassi di questo particolare reindirizzamento è:

```
[n]>parola
```

Ciò fa sì che il file risultante dall'espansione di `parola` venga aperto (o creato nel caso non esista) in scrittura e vi si memorizzino i dati provenienti dal descrittore di file `n`. Nel caso in cui `n` manchi, si assume che si voglia reindirizzare lo standard output, descrittore 1.

È possibile anche aprire un file solo per aggiungere dati, senza perderne il contenuto; per effettuare tale operazione è sufficiente utilizzare l'operatore `>>`, la cui sintassi è analoga al precedente:

```
[n]>>parola
```

Abbiamo visto che un processo possiede due canali di output, uno dedicato in generale alla rappresentazione del risultato (lo standard output), l'altro agli errori incorsi durante l'esecuzione del processo (lo standard error). In genere, in un calcolatore i due output vengono mostrati sullo schermo, con il risultato di confonderli entrambi. Tuttavia, esistono casi in cui è necessario doverli separare; questa operazione può essere eseguita facilmente reindirizzando lo standard error, ad esempio:

```
[nico@deepcool nico]$ ls -lh bg?.{jpg,png} non_esisto
ls: non_esisto: No such file or directory
-rw-rw-r--    1 nico     nico         160k Apr 15 01:20 bg1.jpg
-rw-rw-r--    1 nico     nico         249k Apr 15 01:25 bg2.jpg
-rw-rw-r--    1 nico     nico         198k Mar 30 13:47 bg2.png
-rw-rw-r--    1 nico     nico          86k May 14 11:50 bg3.png

[nico@deepcool nico]$ ls -lh bg?.{jpg,png} non_esisto 2>/dev/null
-rw-rw-r--    1 nico     nico         160k Apr 15 01:20 bg1.jpg
-rw-rw-r--    1 nico     nico         249k Apr 15 01:25 bg2.jpg
-rw-rw-r--    1 nico     nico         198k Mar 30 13:47 bg2.png
-rw-rw-r--    1 nico     nico          86k May 14 11:50 bg3.png
```

Come possiamo vedere, la prima esecuzione del comando `ls -lh bg?.{jpg,png} non_esisto 2>/dev/null`[2] contiene anche un messaggio di errore (inviato sullo standard error) che ci avverte del fatto che il file `non_esisto` non si trova nella directory corrente. Nella seconda esecuzione, aggiungiamo anche un `... 2>/dev/null`, ciò significa che tutto ciò che verra inviato sullo standard error (descrittore di file 2) sarà rigirato nella nostra pattumiera, `/dev/null`.

FIXME: completare.

4.3.5 Duplicare un descrittore di file

Bash consente di duplicare i descrittori di file in modo da organizzare il più funzionalmente possibile i propri script. Ad esempio, può essere necessario assegnare temporaneamente un descrittore aggiuntivo allo standard input, in modo da facilitare l'elaborazione dei dati dopo complesse operazioni di reindirizzamento.

FIXME: Completare.

4.4 Here document

Un *Here document* (Nel seguito sarà abbreviato con *HD*, che non sta per *hard disk*!) è un tipo particolare di reindirizzamento dell'I/O che consente alla shell di leggere delle righe di testo fino ad una parola delimitatrice e trasformarle in standard input di un comando.

La sintassi di questa speciale caratteristica è la seguente:

```
comando <<[-]PAROLA
```

[2] `bg?.{jpg,png}`? Cosa vuol dire? È indetto un concorso a premi per chi indovina!

```
...
(CORPO dell'HD)
...
DELIMITATORE
```

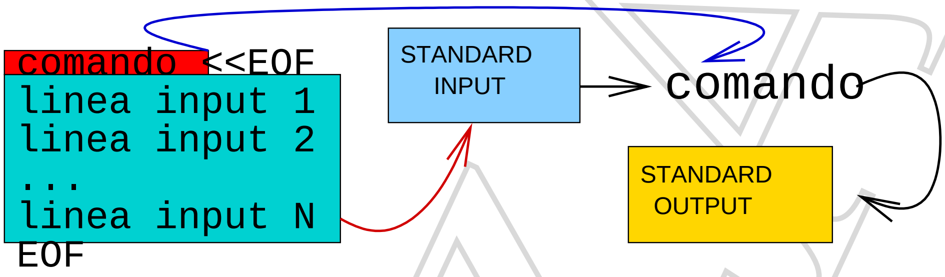

Figura 4.2: Schematizzazione di un *here document* (EOF è il delimitatore).

Su PAROLA non viene fatta alcun tipo di espansione, l'unica cosa da tenere a mente è il fatto che, qualora contenesse un qualsiasi carattere quotato (catattere preceduto da un backslash, \), allora DELIMITATORE sarebbe il risultato della rimozione dei *backslash* da PAROLA e tutte le righe comprese non sarebbero soggette ad alcuna espansione. Se, invece, PAROLA non fosse quotata, sarebbe effettuata espansione di parametri, comandi ed aritmetica su tutte le linee contenute nell'HD.

Il "-" facoltativo serve a rendere più "*graziosa*" la vista di un HD, in questo modo, infatti, le righe possono essere indentate rispeto all'inizio di una riga di un carattere TAB, la sinstassi pertanto diverrebbe:

```
comando <<-PAROLA
        ...
        (CORPO dell'HD)
        ...
        DELIMITATORE
```

Ciò è possibile poiché l'operatore di redirezione <<- istruisce la shell a rimuovere tutti i caratteri TAB che precedono le righe dell'HD e del DELIMITATORE.

Vediamo come applicare quanto appreso con un esempio che fa uso di alcuni semplici comandi previsti dal protocollo *SMTP* (*Simple Mail Transfer Protocol*) e del comando nc (netcat).

Esempio 4.4.1

```
#!/bin/bash
#
# Spedire e-mail con uno shell script
EXIT_SUCCESS=0
EXIT_FAILURE=1
```

```
# indirizzo del server di posta
SMTP_SERVER="localhost"

# percorso completo dell'eseguibile di nc
NC="$(which nc)"

# indirizzo del mittente
MITTENTE="nico@localhost"

utilizzo () {

    echo "$(basename $0) <destinatario> \"<testo>\" \"<oggetto>\""
    exit $EXIT_FAILURE

}

if [ $# -ne 3 ]; then
    utilizzo
fi

DESTINATARIO=$1
TESTO=$2
OGGETTO=$3


$NC -w 5 "$SMTP_SERVER" "25" &> /dev/null <<EOF
HELO $(hostname)
MAIL FROM: <$MITTENTE>
RCPT TO: <$DESTINATARIO>
DATA
Subject: $OGGETTO
To: $DESTINATARIO
From: $MITTENTE

$TESTO

.
EOF

exit $EXIT_SUCCESS
```

Come funziona:

Innanzitutto, vediamo come lanciare lo script.

```
[nico@deepcool nico]$ ./mail.sh nico@deepcool "ecco una e-mail inviata
> con uno shell script" "e-mail con shell script"
[nico@deepcool nico]$
```

Come mostrato nella funzione `utilizzo()`, lo script richiede **tre parametri** per essere eseguito correttamente, rispettivamente: indirizzo e-mail del destinatario, breve testo racchiuso tra virgolette in modo da poterlo estendere su più linee, oggetto della e-mail, anch'esso racchiuso tra virgolette.

Passiamo ad analizzare il codice. Oltre alle solite variabili per definire i valori di ritorno, notiamo subito la presenza di alcune variabili per "configurare" lo script: `$SMTP_SERVER`, `$NC` e `$MITTENTE`. Il significato di queste variabili è, rispettivamente: indirizzo del server smtp da utilizzare, percorso completo dell'eseguibile di netcat (tramite l'utility `which`) e indirizzo e-mail del mittente.

Successvamente, lo script memorizza i parametri posizionali forniti con $1, $2 e $3 nelle variabili `$DESTINATARIO`, `$TESTO` e `$OGGETTO` per maneggiarli più comodamente. Arrivati a questo punto, iniziano le novità. La riga `$NC -w 5 ''$SMTP_SERVER'' ''25'' &> /dev/null <<EOF` identifica un HD in cui tutte le righe comprese tra `<<EOF` ed `EOF` sono trasformate in standard input del comando `$NC -w 5 ''$SMTP_SERVER'' ''25'' &> /dev/null`.

Le righe dell'HD sono comandi standard del protocollo *SMTP*[3]; in particolare, la riga `HELO $(hostname)` è una stringa di "presentazione" che invia al server di posta il nostro hostname, ottenuto tramite il comando `hostname`.

4.5 Opzioni passate ad uno script

Nella sezione precedente abbiamo analizzato un semplice script che ci consentiva di inviare e-mail. Lo script aveva il difetto di dover esser lanciato con una sintassi molto rigida, infatti occorreva richiamarlo fornendo esattamente tre parametri in un ben preciso ordine. Questa restrizione fastidiosa e limitativa potrebbe essere evitata se si potessero usare le opzioni come nei programmi compilati (Ad esempio `ls`, `rm`, `tar` e molti molti altri).

Ancora una volta, *Bash* ci viene in aiuto (Anche altre shell lo fanno), mettendoci a disposizione il comando interno `getopts`.

Andiamo con ordine ed introduciamo alcune peculirità; utilizzando `getopts` all'interno di uno script, vengono implicitamente create le variabili `$OPTIND` (L'indice dell'argomento in esame, che non viene reimpostato automaticamente) e `$OPTARG` (il contenuto del facoltativo argomento di un'opzione). Vediamo ora un quadro che riassume la sintassi del comando:

```
getopts STRINGA_DELLE_OPZIONI NOME
```

`STRINGA_DELLE_OPZIONI` rappresenta la sequenza di opzioni che il nostro script deve avere; ad esempio, se volessimo fornire allo script le opzioni `-a`, `-b` e `-c`, la seguenza sarebbe `''abc''`. Se volessimo anche far in modo che l'opzione `-b` accetti un argomento, allora dovremmo cambiare la sequenza delle lettere, aggiungendo `:` dopo la lettera `b`, otterremo

[3] Per maggiori informazioni sul protocollo SMTP, consultate la rfc 821

dunque `‘‘ab:c’’`. Da qui impariamo una regola generale, ovvero, per informare getopts del fatto che una particolare opzione accetta un argomento, occorre far seguire la lettera che la identifica nella STRINGA_DELLE_OPZIONI da `:`. Porre `:` all'inizio di una stringa equivale a sopprimere ogni messaggio di errore relativo a getopts[4].

NOME, invece, rappresenta la variabile in cui salvare la lettera dell'opzione in esame. Qualora getopts incontrasse un'opzione non presente in STRINGA_DELLE_OPZIONI, inserirebbe in NOME il carattere `?`.

Dovrebbe esser ormai chiaro che getopts analizza le opzioni una alla volta, dunque per analizzare i valori passati occorre utilizzarla insieme con un ciclo. Tra quelli studiati nella sezione 3.1, il ciclo while è senz'altro il più indicato. Non finisce qui! A seconda dell'opzione esaminata occorre prendere decisioni differenti, quindi dobbiamo effettuare una selezione, in questo caso, case è l'istruzione più appropriata. Vediamo dunque come combinare le cose nel caso:

```
...
while getopts ":ab:c" OPZIONE; do
    case $OPZIONE in
        a)
            fai_qualcosa_con_a
            ;;
        b)
            VARIABILE_PER_b=$OPTARG
            fai_qualcosa_con_b
            ;;
        c)
            fai_qualcosa_con_c
            ;;
        ?)
            echo "Opzione non valida"
            ;;
        *)
            echo "C’è stato qualche errore!"
            ;;
    esac
done

shift $(( $OPTIND - 1 ))
```

Vediamo dunque come modificare l'esempio della sezione precedente in modo da renderlo più utilizzabile.

Esempio 4.5.1

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

[4] Un sistema per ottenere lo stesso risultato è quello di impostare a zero la variabile $OTPERR.

```
#
# Spedire e-mail con uno shell script (2)
EXIT_SUCCESS=0
EXIT_FAILURE=1

# indirizzo del server di posta
SMTP_SERVER="localhost"

# percorso completo dell'eseguibile di nc
NC="$(which nc)"

# l'indirizzo del mittente
MITTENTE="nico@localhost"

utilizzo () {

    echo "$(basename $0) -d <destinatario> -t <testo> -o <oggetto>"
    exit $EXIT_FAILURE

}

while getopts ":d:t:o:" OPZIONE; do
    case $OPZIONE in
        d)
            DESTINATARIO=$OPTARG
            ;;
        t)
            TESTO=$OPTARG
            ;;
        o)
            OGGETTO=$OPTARG
            ;;
        *)
            utilizzo
            ;;
    esac
done

shift $(( $OPTIND - 1 ))

if [ -z "$DESTINATARIO" -o -z "$TESTO" -o -z "$OGGETTO" ]; then
    utilizzo
fi

$NC -w 5 "$SMTP_SERVER" "25" &> /dev/null <<EOF
HELO $(hostname)
MAIL FROM: <$MITTENTE>
```

```
RCPT TO: <$DESTINATARIO>
DATA
Subject: $OGGETTO
To: $DESTINATARIO
From: $MITTENTE

$TESTO

.
EOF

exit $EXIT_SUCCESS
```

Come funziona:

L'unica novità rispetto alla precedente versione è, come era da aspettarsi, la presenza di `getopts`. Abbiamo passato a `getopts` la stringa `'':d:t:o:''`, in questo modo, grazie al `:` iniziale abbiamo soppresso ogni messaggio di errore, inoltre abbiamo istruito il comando ad analizzare i parametri posizionali alla ricerca delle opzioni `-d` (destinatario), `-t` (testo) e `-o` (oggetto), ognuna delle quali necessita di un argomento. Attraverso l'istruzione case assegnamo l'argomento di una opzione ad una specifica variabile, in modo da poterlo riutilizzare; inoltre, dato che non ci interessa essere molto specifici riguardo ad un eventuale errore, abbiamo tralasciato il caso `?)`, dato che questo può essere inglobato in quello di default `*)`.

4.6 . (source)

Alle volte potrebbe rendersi necessario includere in uno script del testo presente in un altro file; questo testo potrebbe essere del codice e quindi dovrebbe anche essere interpretato correttamente dalla shell. La questione può essere risolta facilmente utilizzando il comando interno `source` o `.`.

Un tale tipo di inclusione può essere necessario qualora decidessimo di dotare un nostro script di un file di configurazione. Ad esempio nell'esempio precedente esistono delle variabili dipendenti dal calcolatore su cui viene eseguito, sarebbe auspicabile, dunque, avere un sistema per spiegare allo script, ad esempio, quale server di posta utilizzare, oppure quale argomento passare al comando SMTP `HELO`.

Vediamo come affrontare questa evenienza.

Esempio 4.6.1

Innanzi tutto, stabiliamo il percorso del file di configurazione, ad esempio `/etc/bash_mail.conf` e scriviamoci:

```
# Questo è un commento
```

```
# server SMTP da utilizzare
SMTP_SERVER="il_mio_server_di_posta"

# HELO da utilizzare
# Modificare a seconda delle esigenze
HELO="il_mio_host"

# fine di /etc/bash_mail.conf
```

Potremmo anche consentire ad ogni singolo utente di modificare i paramentri di configurazione dello script; per fare ciò, prevediamo anche l'esistenza di un file `$HOME/.bash_mail.conf`

```
# File di configurazione per utente

# server SMTP da utilizzare
# SMTP_SERVER="un_altro_server"

# HELO da utilizzare
# Modificare a seconda delle esigenze
# HELO="un_altro_helo"

# Indirizzo del mittente
MITTENTE="Domenico Delle Side <nicodds@Tiscali.IT>"

# fine di $HOME/.bash_mail.conf
```

Ora riprendiamo l'esempio e modifichiamolo in modo da utilizzare i file di configurazione appena scritti.

```
#!/bin/bash
#
# Spedire e-mail con uno shell script (3)
EXIT_SUCCESS=0
EXIT_FAILURE=1

NC=$(which nc)

if [ -f "/etc/bash_mail.conf" ]; then
    . /etc/bash_mail.conf
# oppure
# source /etc/bash_mail.conf
else
    SMTP_SERVER="localhost"
    HELO=$HOSTNAME
fi
```

```
if [ -f "$HOME/.bash_mail.conf" ]; then
    . $HOME/.bash_mail.conf
else
    MITTENTE="<${USER}@${HOSTNAME}>"
fi

utilizzo () {

    echo "$(basename $0) -d <destinatario> -t <testo> -o <oggetto>"
    exit $EXIT_FAILURE

}

while getopts ":d:t:o:" OPZIONE; do
    case $OPZIONE in
        d)
            DESTINATARIO=$OPTARG
            ;;
        t)
            TESTO=$OPTARG
            ;;
        o)
            OGGETTO=$OPTARG
            ;;
        *)
            utilizzo
            ;;
    esac
done

shift $(( $OPTIND - 1 ))

if [ -z "$DESTINATARIO" -o -z "$TESTO" -o -z "$OGGETTO" ]; then
    utilizzo
fi

$NC -w 5 "$SMTP_SERVER" "25" &> /dev/null <<EOF
HELO $HELO
MAIL FROM: $MITTENTE
RCPT TO: <$DESTINATARIO>
DATA
Subject: $OGGETTO
To: $DESTINATARIO
From: $MITTENTE

$TESTO
```

```
.
EOF

exit $EXIT_SUCCESS
```

Come funziona:

Lo script in sè non è radicalmente cambiato, rispetto alla versione precedente abbiamo solo aggiunto una nuova variabile (`$HELO`) e modificato il codice per consentire l'inclusione dei file di configurazione. Come prima cosa, lo script verifica la presenza del file di configurazione generico `/etc/bash_mail.conf`, se presente, lo include, altrimenti assegna dei valori di default alle variabili `$SMTP_SERVER` e `$HELO`.

Successivamente viene verificata la presenza del file di configurazione dell'utente che lancia lo script, `$HOME/.bash_mail.conf` (`$HOME` viene espansa dalla shell nella home directory dell'utente che esegue lo script). Qualora il file fosse presente, ne sarebbe incluso il contenuto, in questo modo le variabili generiche eventualmente impostate con l'inclusione precedente verrebbero reimpostate ai valori presenti nel file dell'utente. Se invece il file fosse assente, lo script si limiterebbe ad impostare la variabile `$MITTENTE`.

Per il resto, lo script è analogo all'esempio della sezione precendente.

4.7 Subshell

4.8 `trap`

`trap` è un comando interno alla shell che consente di definire un comando da eseguire allorché lo script in esecuzione riceva un **segnale**.

In breve, un **segnale** è un evento generato da un sistema *Unix* al verificarsi di una particolare condizione. Un segnale è in genere utilizzato dal sistema operativo per dialogare con i processi; un processo che riceve un segnale, ad esempio, può a sua volta decidere un'azione da intraprendere.

Ritorniamo a `trap` e analizziamone la sintassi:

```
trap [-lp] [AZIONE] [SEGNALE]
```

Procediamo con ordine, le opzioni facoltative (`-l`, `-p`) fanno agire `trap` in maniera particolare; l'opzione `-l` viene passata da sola a `trap` e consente di ottenere la lista di tutti i segnali supportati, `-p`, invece, consente di vedere le azioni associate ad ogni segnale, o, qualora fossero specificati dei segnali, solo quelle associate a questi. Ad esempio, il comando `trap -l` ci fornisce il risultato:

```
[nico@deepcool nico]$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
```

```
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

Il parametro `AZIONE` (anch'esso opzionale) serve a specificare, come chiarisce il nome stesso, l'azione da intraprendere al ricevere del segnale `SEGNALE` (opzionale), in genere questa è una lista di comandi da eseguire. Il segnale `SEGNALE` può essere specificato sia tramite una delle costanti simboliche definite in `<signal.h>`[5], sia attraverso il numero che identifica il segnale. In tabella 4.1 è presente una breve lista di segnali standard.

Segnale	Numero	Descrizione
SIGHUP	1	Hangup, il processo che riceve il segnale viene terminato e fatto ripartire
SIGINT	2	Interrupt, il procevo che riceve il segnale viene terminato (CTRL-C)
SIGQUIT	3	Quit (CTRL-\)
SIGABRT	6	Abort, inviato ad un processo in caso di gravi errori in esecuzione
SIGALRM	14	Alarm, inviato in caso di time-out
SIGTERM	15	Terminate, inviato dal sistema in caso di spegnimento

Tabella 4.1: Alcuni segnali più importanti definiti dallo standard *X/Open*

Ad esempio, se volessimo far sì che nel nostro terminale venga stampato un messaggio ogni volta che si riceva un segnale di interrupt, basterebbe scrivere:

```
trap 'echo -e "\nQuesto un messaggio impostato con trap (SIGINT)"' SIGINT
```

Dopo averlo eseguito, infatti, ad ogni pressione della combinazione di tasti CTRL-C otteniamo il messaggio scritto:

```
[nico@deepcool nico]$ trap 'echo -e "\nQuesto un messaggio impostato
> con trap (SIGINT)"' SIGINT
[nico@deepcool nico]$ (CTRL-C)
Questo un messaggio impostato
con trap (SIGINT)
```

[5] Anche nei file header inclusi.

```
[nico@deepcool nico]$ (CTRL-C)
Questo un messaggio impostato
con trap (SIGINT)

[nico@deepcool nico]$ (CTRL-C)
Questo un messaggio impostato
con trap (SIGINT)

[nico@deepcool nico]$ (CTRL-C)
Questo un messaggio impostato
con trap (SIGINT)
```

Se ora volessimo tornare alla situzione iniziale, basterebbe digitare il comando:

```
trap SIGINT
```

Ciò riporterebbe l'azione da intraprendere al ricevimento di un interrupt al default della shell.

Esempio 4.8.1

```
#!/bin/bash
#
# Uso di trap

EXIT_SUCCESS=0

TMPFILE=$HOME/.tmpfile.$$

trap 'rm -f $TMPFILE' SIGINT

echo "Creo il file $TMPFILE..."
touch $TMPFILE
echo "Premi CRTL-C per uscire"

while [ -f $TMPFILE ]; do
    echo "$TMPFILE esiste ancora"
    sleep 1;
done

echo "$TMPFILE non esiste più"

exit $EXIT_SUCCESS
```

Come funziona:

All'inizio dello script definiamo la variabile $TMPFILE e definiamo un azione da compiere su questo al ricevere di un segnale di interrupt: `trap 'rm -f $TMPFILE' SIGINT`. Successivamente, dopo alcuni messaggi di cortesia, creiamo il file temporaneo $TMPFILE con l'utility touch ed entriamo in un ciclo nel quale la condizione è data da `-f $TMPFILE`. Ad ogni iterazione del ciclo viene stampato un messaggio che conferma l'esistenza del file temporaneo e si manda in pausa per 1 secondo l'esecuzione dello script grazie al comando `sleep`.

Alla pressione della combinazione di tasti CTRL-C, il file temporaneo verrà cancellato, dunque la condizione valutata dal ciclo `while` sarà falsa, causandone la fine. Lo script termina stampando un ulteriore messaggio e ritornando alla shell il valore `$EXIT_SUCCESS`.

Capitolo 5

Esempi avanzati

In questo capitolo saranno raccolti degli script più complicati (niente paura, nulla di difficle), vicini alla realtà di un utente e pertanto utili. Eventualmente, si troveranno contributi da parte degli utenti, i quali possono inviare i propri script con un breve commento.

Gli esempi saranno divisi per argomento ed i commenti saranno, per la maggior parte, all'interno del codice stesso.

5.1 Script d'amministrazione

5.2 Utilità

In questa sezione verranno riportati alcuni script utili nell'uso quotidiano.

5.2.1 Esempio 5.2.1: Creare un cgi con la shell

Negli ultimi anni sono state sviluppate nuove tecnologie e nuovi linguaggi per semplificare la vita a chi scrive *cgi* (Common Gateway Interface, volgarmente: pagine web a contenuto dinamico). Creare un cgi con *Bash* può pertanto sembrare una forzatura, dato che questa non è la scelta tecnica migliore, ciò nonostante tratteremo questo argomento perché ricco dal punto di vista didattico.

Per semplicità, tratteremo in dettaglio soltanto il metodo `GET` del protocollo *HTTP*, `POST` sarà accennato verso la fine.

Per poter eseguire con successo lo script, occorrerà avere un server web (ad esempio *Apache*, `http://httpd.apache.org`) funzionante, configurato per supportare l'esecuzione di cgi.

Utilizzando il metodo `GET`, le "coppie" `variabile=valore` vengono passate attraverso l'url. Ad esempio spesso vediamo url del tipo :
`http://www.example.com/cgi-bin/script?var1=ciao&var2=bau`
Vediamo dunque che le coppie `variabili=valore` sono separate l'una dall'altra attraverso

il carattere &, mentre sono separate dal percorso dello script attraverso ?. Il server web

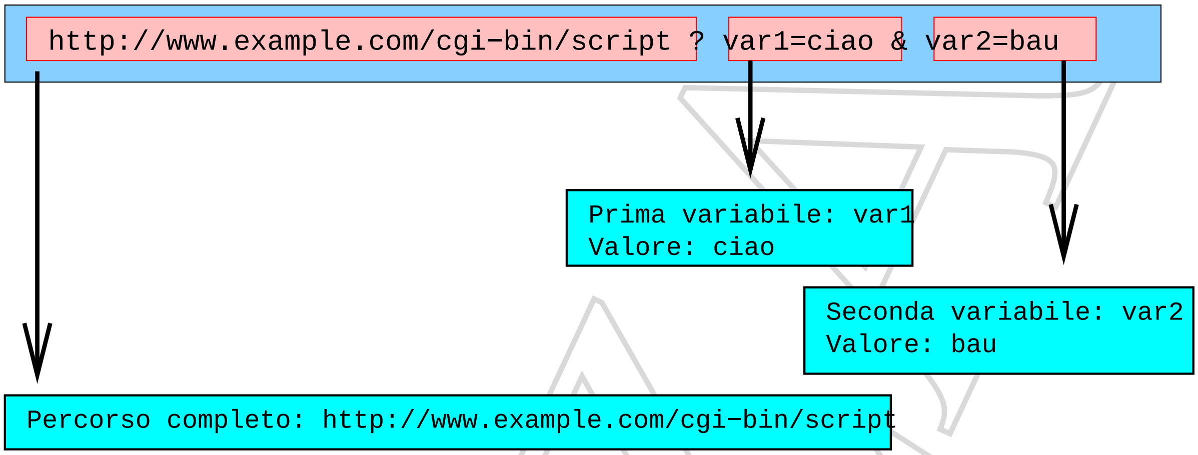

Figura 5.1: Url con il metodo GET

mette a disposizione la stringa a destra del carattere ? in un'apposita variabile d'ambiente, `$QUERY_STRING`. Per ottenere le coppie `variabile=valore` sarà dunque sufficiente analizzare il contenuto di tale stringa.

Vediamo nella pratica come realizzare lo script.

```
#!/bin/bash
#
# Esempio di cgi con bash

EXIT_SUCCESS=0

# dichiariamo un vettore in cui salvare i valori delle variabili
# passate con GET

declare -a QUERY


# Una funzione attraverso la quale impostare la parte di testa
# della pagina. La stringa "Content-type: text/html" serve ad istruire
# il browser ad interpretare quanto segue come html. Se avessimo,
# invece scritto "Content-type: text/plain" otterremmo la pagina
# senza che il codice html venga interpretato.
#
# La funzione accetta un argomento, il titolo della pagina.

sopra () {

    cat <<EOF
Content-type: text/html
```

```
<HTML>
<HEAD>
  <TITLE>$1</TITLE>
</HEAD>
<BODY>
<!-- fine parte superiore -->

EOF

}


# Con questa funzione, invece, definiamo gli elementi comuni della
# parte inferiore della pagina, in modo da chiuderla correttamente.

sotto () {

    cat <<EOF

<!-- inizio parte inferiore -->
</BODY>
</HTML>
EOF

}


# La funzione che segue è il "cuore" dello script, grazie a questa
# possiamo analizzare il contenuto di $QUERY_STRING, ottenere il
# valore di ogni variabile e salvarlo in un elemento del vettore
# $QUERY.
#
# Si noti il particolare modo in cui abbiamo iterato con il ciclo
# while; abbiamo dovuto "sprecare" il primo elemento del vettore. Se
# non avessimo agito in questo modo, infatti, la condizione valutata
# dal ciclo sarebbe stata falsa alla prima esecuzione e non avremmo
# mai ottenuto le variabili che ci interessano. Questo artificio
# simula in qualche modo il ciclo "do ... while" presente in altri
# linguaggi quali ad esempio il C.

analizza_query () {

    i=0
    QUERY[0]="OK"

    while [ -n "${QUERY[${i}]}" ]; do
        # Aumentiamo subito il valore di $i
```

```
        ((i=$i+1))
        # Otteniamo il valore delle variabili
        QUERY[${i}]="$(echo $QUERY_STRING | cut -d \& -f ${i} \
            | cut -d = -f 2 | sed 's/+/ /g')"
    done

}


# Questa funzione ci consente di ottenere una form attraverso la quale
# ottenere un input dall'esterno.

prima_pagina () {

    cat <<EOF
<P>
  Inserisci nome e cognome:
</P>

<FORM method="GET" action="$(basename $0)">
  <P>
    <B>Nome</B>:<BR> <INPUT type="TEXT" name="nome">
  </P>
  <P>
    <B>Cognome</B>:<BR> <INPUT type="TEXT" name="cognome">
  </P>
  <INPUT type="SUBMIT">
</FORM>
EOF

}


# Con questa funzione mostriamo i valori ottenuti dall'esterno.

seconda_pagina () {
    cat <<EOF
<H2>
  Ciao ${QUERY[1]} ${QUERY[2]}!
</H2>

<P>
  Hai visto quanto è divertente fare cgi con bash?
</P>

<P>
  <A href="$(basename $0)">Torna alla pagina precedente</A>
```

```
</P>
EOF

}

# FINE delle funzioni

analizza_query

if [ -z "${QUERY[1]}" ]; then
    sopra "Inserisci nome e cognome"
    prima_pagina
else
    sopra "Risultato"
    seconda_pagina
fi

sotto

exit $EXIT_SUCCESS
```

Come funziona:

Analizzeremo principalmente parte dellla funzione `analizza_query()` ed il codice al di sotto di `# FINE delle funzioni`.

Procediamo con ordine ed occupiamoci in particolare della stringa di codice

```
QUERY[${i}]="$(echo $QUERY_STRING | cut -d \& -f ${i} \
            | cut -d = -f 2 | sed 's/+/ /g')"
```

presente all'interto di `analizza_query()`. `QUERY[${i}]` è un elemento di un vettore, mentre la stringa di desta è il risultato di una serie di pipe di alcuni comandi,

```
echo $QUERY_STRING | cut -d \& -f ${i} \
            | cut -d = -f 2 | sed 's/+/ /g'}
```

Analizziamoli uno ad uno. Il primo, `echo $QUERY_STRING`, stampa sullo standard output il contenuto della variabile `$QUERY_STRING` che, grazie alla pipe, diventa standard input di `cut -d \& -f ${i}`. In generale, `cut` è un comando che serve ad eseguire operazioni avanzate di estrazione di testo da una stringa; in particolare, con la sintassi utilizzata, diciamo a `cut` di dividere la stringa in campi (Utilizzando come separatore &, che deve essere preceduto da un backslash per evitare che la shell lo interpreti come istruzione) ed estrarne quello di posto `${i}`. In questo modo, otteniamo la coppia `variabile=valore` di posto `${i}`.

Il secondo `cut` è del tutto analogo al primo, tuttavia questa volta si usa come separatore di campi il carattere = (Non ha bisogno di alcun escape) e se ne ritorna il campo di posto 2; così facendo, otteniamo il valore della variabile in esame.

In ultimo, dobbiamo tener presente il fatto che ogni spazio presente nel valore di una variabile viene sostituito con un +, pertanto, quando ritorniamo i valori dobbiamo ricordarci di risostituire ogni occorrenza di + con uno spazio. Questa operazione viene effettuata attraverso il comando `sed 's/+/ /g'`.

Passiamo ora ad analizzare il resto. Dopo aver dichiarato tutte le funzioni necessarie, inizia la parte dello script che prende le decisioni. Innanzi tutto, lanciamo la funzione `analizza_query()` e decidiamo cosa fare a seconda del suo operato, infatti controlliamo l'elemento `${QUERY[1]}`, se il suo valore non è impostato, allora lo script è stato richiamato senza passare alcun paramentro nell'url, dunque eseguiamo le funzioni `sopra()` (Passandole come argomento la stringa "Inserisci nome e cognome") e `prima_pagina()` per raccogliere i dati.

Se, invece, il valore di `${QUERY[1]}` fosse impostato, allora lo script sarebbe stato richiamato passandogli dei parametri, quindi si richiamerebbero le funzioni `sopra()` (Argomento "Risultato") e `seconda_pagina()` in modo da mostrare i valori raccolti.

In ultimo, viene richiamata la funzione `sotto()` per chiudere correttamente il codice html della pagina e si ritorna un valore di successo.

5.3 One-line

Con *one-line* intendiamo dei semplici script che si sviluppano tutti su una sola linea di codice, utili spesso per portare a termine in un batter d'occhio i lavori sporchi.

5.3.1 Esempio 5.3.1: terminare un processo per nome

Può capitare, ad esempio, di avere a che fare con programmi che non rispondono più ai comandi (raramente) ed occorre dunque terminarli. Andare alla ricerca del PID (Process ID) del processo per poi mandare al programma un segnale di `KILL` può essere fastidioso, sarebbe più bello se, magari, potessimo fare tutto fornendo il solo nome del programma. Siamo fortunati, leggendo la pagina manuale di `ps`, possiamo notare l'esistenza dell'opzione `-C` (Controllate!) che fa proprio al caso nostro. Vediamo, dunque, come fare.

```
$ kill -9 $(ps -C nome_comando -o pid=)
```

o

```
$ kill -KILL $(ps -C nome_comando -o pid=)
```

Come funziona:

Analizziamo passo passo ciò che abbiamo scritto. Il comando `kill` serve ad inviare segnali ai processi; nel nostro caso, stiamo inviando il segnale di `KILL` (terminazione) attraverso l'opzione `-9` [1]. Tale comando necessita come input del PID del processo in questione, per questo motivo

[1] Per una lista completa dei segnali supportati dal comando, si digiti in un terminale `kill -l`

abbiamo utilizzato l'espanzione di comandi `$( ... )` su `ps -C nome_comando -o pid=`, con le opzioni specificate, infatti, `ps` dà come output il pid[2] del processo lanciato da `nome_comando`[3]. Per concludere, se volessimo terminare il processo lanciato dal programma `gabber`[4] dovremmo digitare i comandi:

```
$ kill -9 $(ps -C gabber -o pid=)
```

[2]Alcuni sistemi mettono a disposizione il comando `pidof` che consente di ottenere il PID di un processo

[3]**Esercizio**: capire perchè avviene quanto detto (basta leggere la manpage).

[4]*Gabber* è il client *Jabber* per l'ambiente desktop *GNOME*. Per maggiori informazioni, `http://gabber.sf.net`

Appendice A

GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.

- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified

Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or any later version applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliografia

[1] Richard Stones, Neil Matthew. *Beginning Linux Programming*. Wrox Press, 1999.

[2] Mendel Cooper. *Advanced Bash Scripting Guide*. 2001.

[3] Brian Fox, Chet Ramey. *Bash Reference Manual*. Free Software Foundation.

[4] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Matematica Discreta* Ulrico Hoepli Editore, Milano 1992.

Indice analitico

P

R

S

T

V